



# Lesson 10

19.04.2021

```
public class EX1 {  
    public static void main(String[] args) {  
        List<Integer> list = new ArrayList<>();  
        list.add(1);  
        list.add(2);  
        list.add(3);  
  
        for (Integer i : list) {  
            System.out.print(i + " ");  
            break;  
        }  
    }  
}
```



```
public class EX2 {  
    public static void main(String[] args) {  
        int x = 0;  
        String s = null;  
        if (x == s) System.out.println("S");  
        else System.out.println("F");  
    }  
}
```

```
public class EX3 {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<String>();  
        list.add("1");  
        list.add("2");  
        list.add(3);  
  
        for (String s : list) {  
            System.out.print(s + " ");  
            break;  
        }  
    }  
}
```

```
| public class EX4 {  
|     public static void main(String[] args) {  
|         List<String> list1 = new ArrayList<>();  
|         list1.add("one");  
|         List<String> list2 = new ArrayList<>();  
|         list2.add("one");  
  
|         if (list1 == list2) System.out.println(1);  
|         else if (list1.equals(list2)) System.out.println(2);  
|         else System.out.println(3);  
|     }  
| }
```

```
public class EX5 {  
    public static void main(String[] args) {  
        String o = "-";  
        switch ("FRED".toLowerCase().substring(1, 3)) {  
            case "yellow":  
                o += "y";  
            case "red":  
                o += "f";  
            case "green":  
                o += "g";  
        }  
        System.out.println(o);  
    }  
}
```

# Iterator

Итератор способен поочередно обойти все элементы в коллекции. При этом он позволяет это сделать без вникания во внутреннюю структуру и устройство коллекций.

## Итератор

Iterator

```
Iterator.java
Inherited members (Ctrl+F12) Anonymous Classes (Ctrl+I) Lambdas (Ctrl+L)
I Iterator
m forEachRemaining(Consumer<? super E>): void
m hasNext(): boolean
m next(): E
m remove(): void
```



Даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

Методы, которые должен имплементировать Iterator:

**boolean hasNext()** — если в итерируемом объекте (пока что это Collection) остались еще значение — метод вернет true, если значения кончились false.

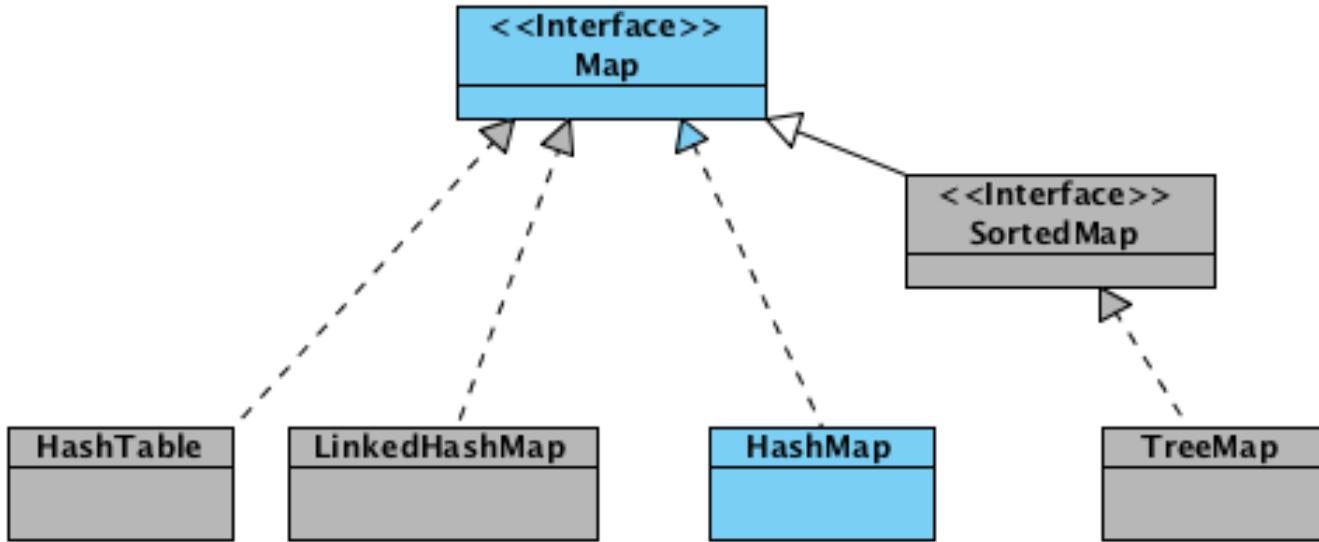
**E next()** — возвращает следующий элемент коллекции (объекта). Если элементов больше нет (не было проверки hasNext(), а мы вызвали next(), достигнув конца коллекции), метод бросит NoSuchElementException.

**void remove()** — удалит элемент, который был в последний раз получен методом next(). Метод может бросить:

*UnsupportedOperationException*, если данный итератор не поддерживает метод remove() (в случае с read-only коллекциями, например)

*IllegalStateException*, если метод next() еще не был вызван, или если remove() уже был вызван после последнего вызова next().

# HashMap



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.



## Создание объекта

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

Новоявленный объект hashmap, содержит ряд свойств:

**table** — Массив типа **Entry[]**, который является хранилищем ссылок на списки (цепочки) значений;

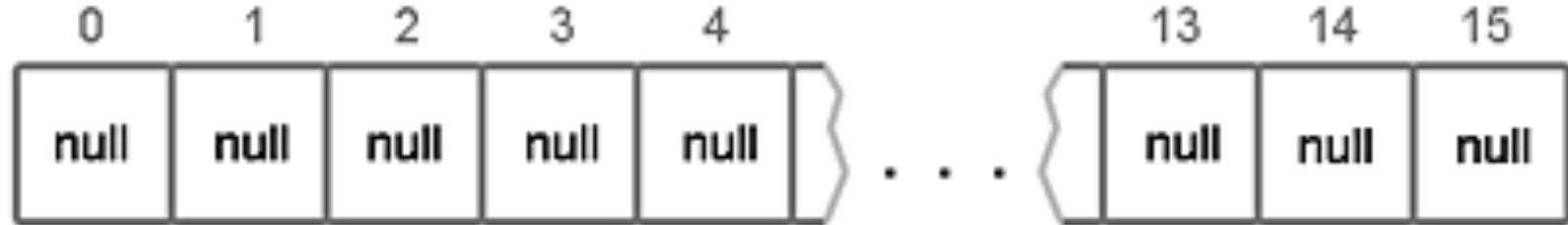
**loadFactor** — Коэффициент загрузки. Значение по умолчанию 0.75 является хорошим компромиссом между временем доступа и объемом хранимых данных;

**threshold** — Предельное количество элементов, при достижении которого, размер хэш-таблицы увеличивается вдвое. Рассчитывается по формуле **(capacity \* loadFactor)**;

**size** — Количество элементов HashMap-а;

## Операции с Map

1. **put(K key, V value)** - добавляет элемент в карту;
2. **get(Object key)** - ищет значение по его ключу;
3. **remove(Object key)** - удаляет значение по его ключу;
4. **containsKey(Object key)** - спрашивает, есть ли в карте заданный ключ;
5. **containsValue(Object value)** - спрашивает есть ли в карте заданное значение;
6. **size()** - возвращает размер карты (количество пар "ключ-значение").



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).

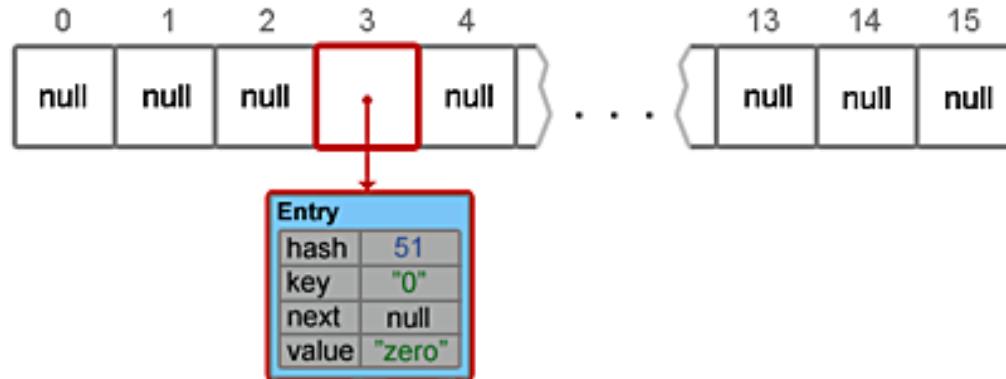


## Добавление элементов

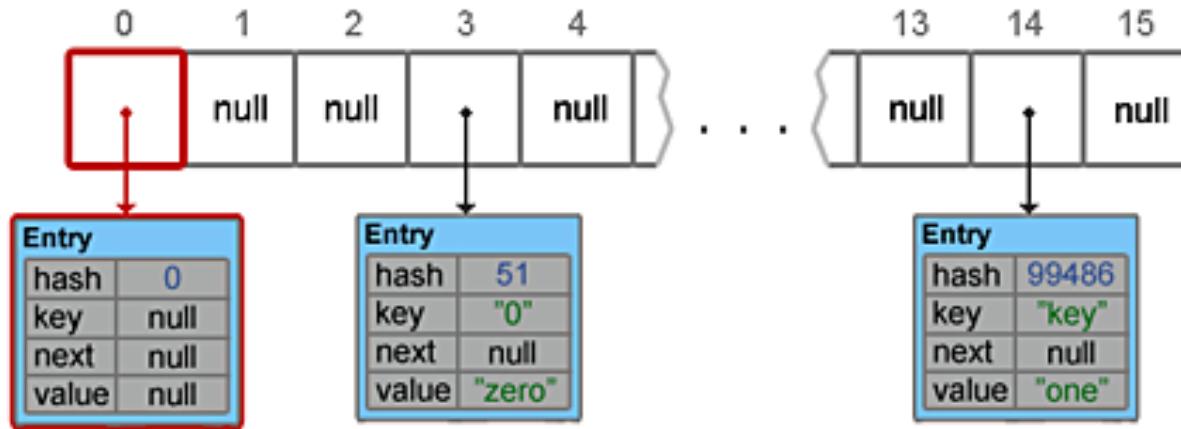
```
hashmap.put("0", "zero");
```

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](#)).
2. Далее генерируется хэш на основе ключа. Для генерации используется метод **hash(hashCode)**, в который передается **key.hashCode()**.
3. С помощью метода **indexFor(hash, tableLength)**, определяется позиция в массиве, куда будет помещен элемент.
4. 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.

5. Если же предыдущий шаг не выявил совпадений, будет вызван метод **addEntry(hash, key, value, index)** для добавления нового элемента.

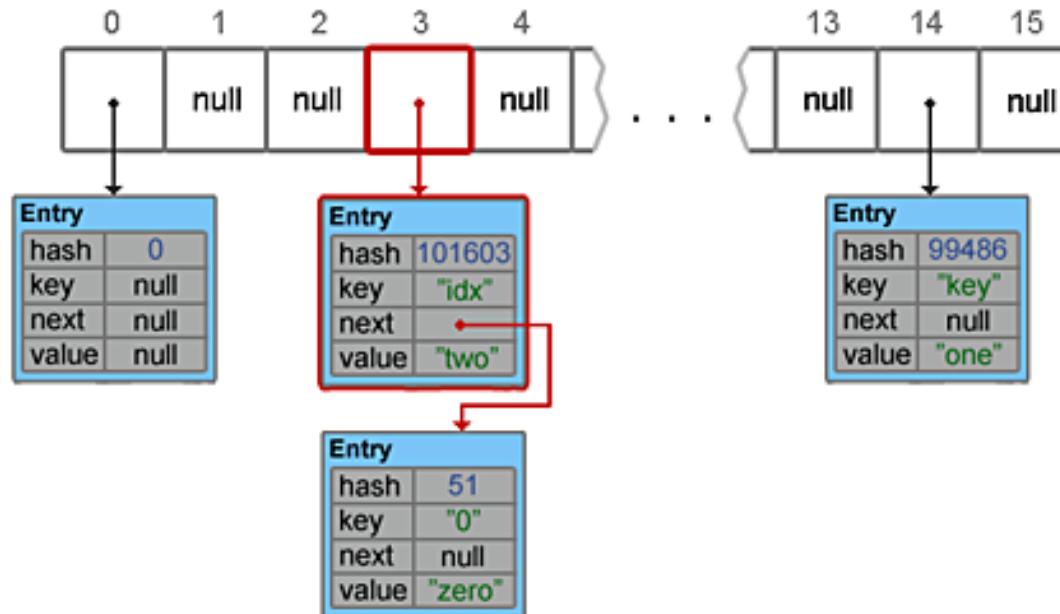


```
hashmap.put(null, null);
```



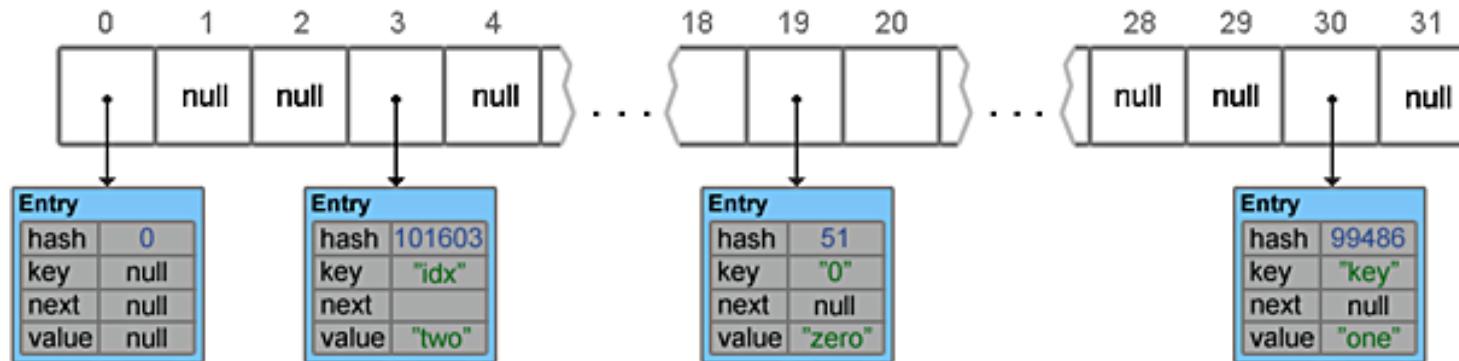
## Коллизия

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



## 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.

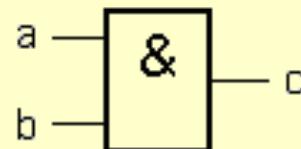


## Итераторы

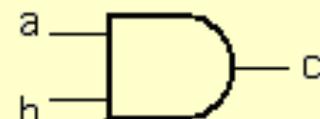
HashMap имеет встроенные итераторы, такие, что вы можете получить список всех ключей **keySet()**, всех значений **values()** или же все пары ключ/значение **entrySet()**. Ниже представлены некоторые варианты для перебора элементов:

```
// 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());
```

```
static int indexFor(int h, int length)
{
    return h & (length - 1);
}
```

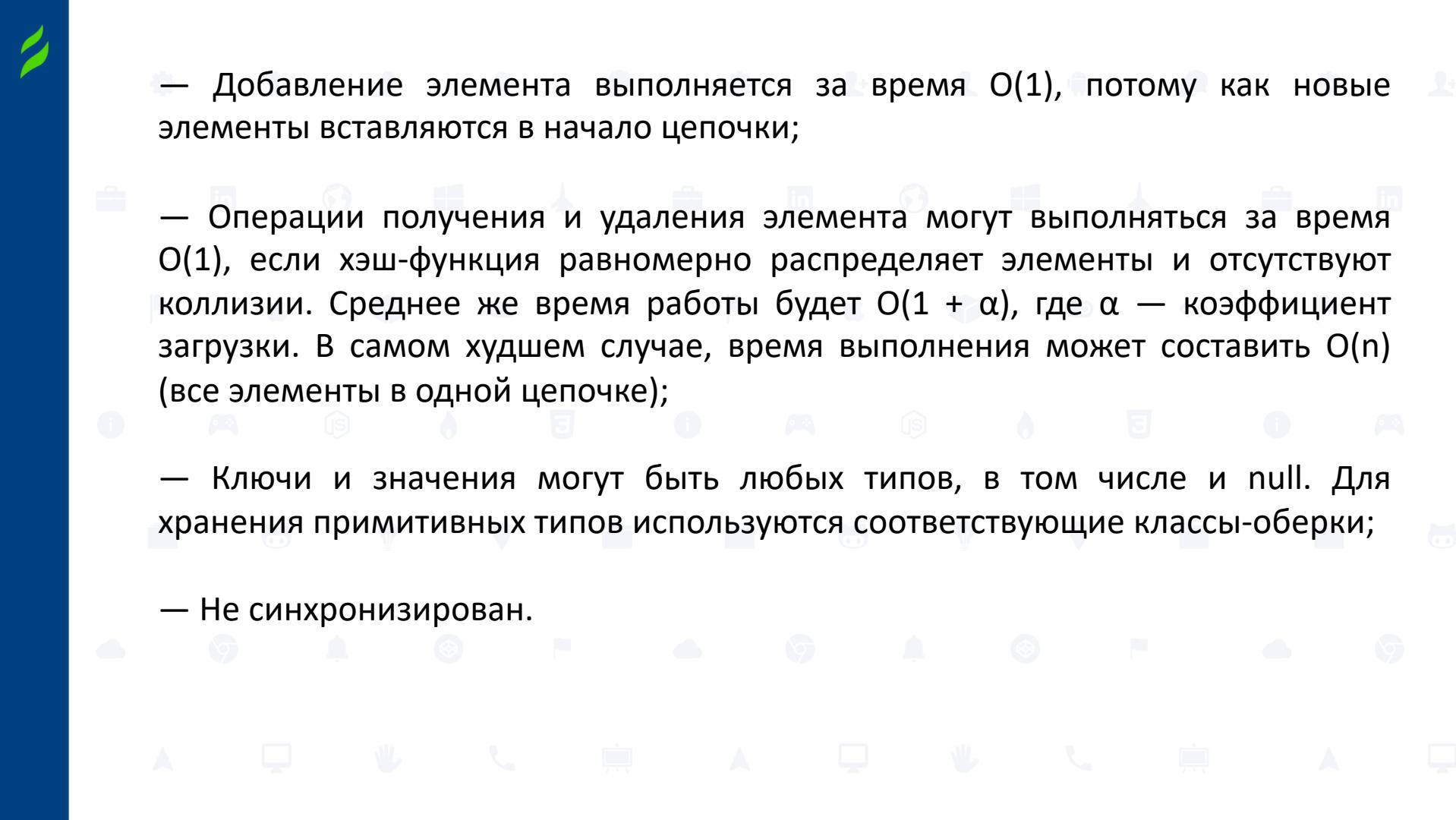


(наш стандарт)



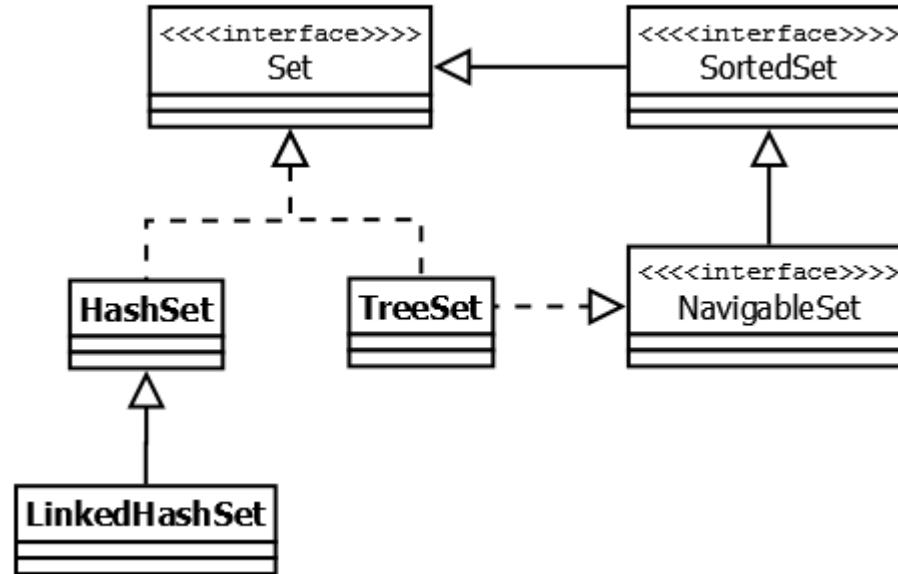
(буржуйский стандарт)

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1



- 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  $O(1 + \alpha)$ , где  $\alpha$  — коэффициент загрузки. В самом худшем случае, время выполнения может составить  $O(n)$  (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.

# HashSet



HashSet — реализация интерфейса Set, базирующаяся на HashMap. Внутри использует объект HashMap для хранения данных. В качестве ключа используется добавляемый элемент, а в качестве значения — объект-пустышка (new Object()). Из-за особенностей реализации порядок элементов не гарантируется при добавлении.

## Операции с множествами

1. **add()** - добавляет элемент в множество
2. **remove()** - удаляет элемент из множества
3. **contains()** - определяет, есть ли элемент в множестве
4. **size()** - возвращает размер множества
5. **clear()** - удаляет все элементы из коллекции
6. **isEmpty()** - возвращает true если множество пустое, и false если там есть хотя бы 1 элемент



## String

У String есть две фундаментальные особенности:

- это `immutable` (неизменный) класс
- это `final` класс

В общем, у класса String не может быть наследников (`final`) и экземпляры класса нельзя изменить после создания (`immutable`).

# StringBuffer и StringBuilder

## Отличие между String, StringBuilder, StringBuffer:

- Классы StringBuffer и StringBuilder в Java используются, когда возникает необходимость сделать много изменений в строке символов.
- В отличие от String, объекты типа StringBuffer и StringBuilder могут быть изменены снова и снова.
- Основное различие между StringBuffer и StringBuilder в Java является то, что методы StringBuilder не являются безопасными для потоков (несинхронизированные).
- Рекомендуется использовать StringBuilder всякий раз, когда это возможно, потому что он быстрее, чем StringBuffer в Java.
- Однако, если необходима безопасность потоков, наилучшим вариантом являются объекты StringBuffer.

Метод **charAt()** возвращает символ в указанной позиции. А **setCharAt()** изменяет символ в указанной позиции

Метод **append()** присоединяет подстроку к строке

Метод **insert()** вставляет подстроку в указанную позицию

Метод **reverse()** используется для инвертирования строки

Метод **delete()** удаляет подстроку, используя указанные позиции

Метод **deleteCharAt()** удаляет символ с указанной позиции

Метод **replace()** заменяет подстроку в указанной позиции другой