Collections and generics

Сергей Товмасян

HashMap – Объекты/equals/hashCode

Требования к реализации equals() и hashCode()
Для поведения equals() и hashCode() существуют некоторые ограничения, которые указаны в документации по Object. В частности, метод equals() должен обладать следующими свойствами:

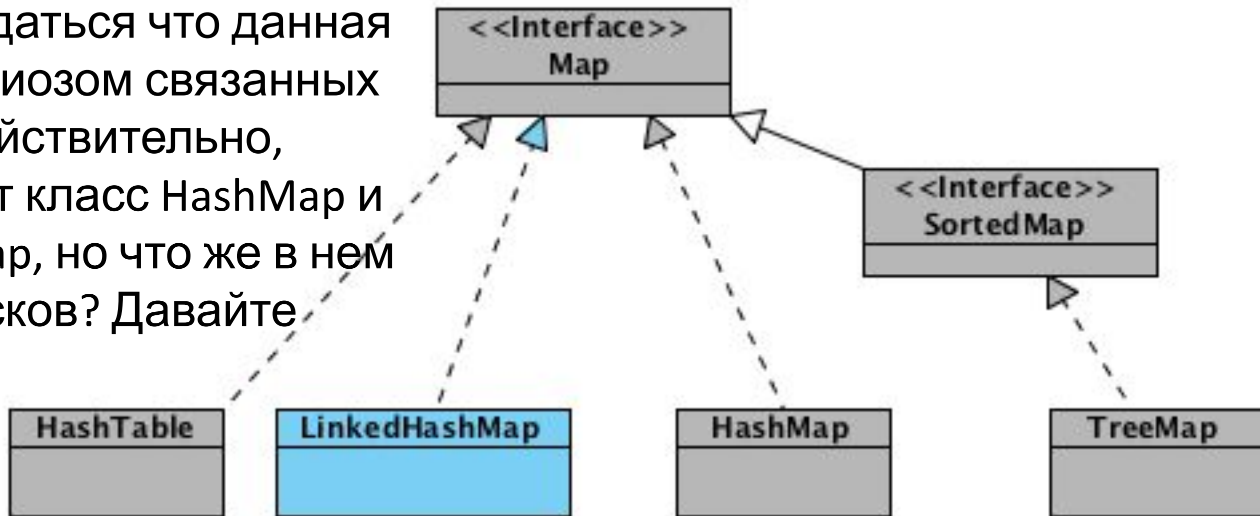
1. **Симметричность**: Для двух ссылок, a и b, a.equals(b) тогда и только тогда, когда b.equals(a)
2. **Рефлексивность**: Для всех ненулевых ссылок, a.equals(a)
3. **Транзитивность**: Если a.equals(b) и b.equals(c), то тогда a.equals(c)
4. **Совместимость с hashCode()**: Два тождественно равных объекта должны иметь одно и то же значение hashCode()

Сгенерим в Idea и подумаем о магических числах.

31 * i == (i << 5) - i

LinkedHashMap

Из названия можно догадаться что данная структура является симбиозом связанных списков и хэш-мапов. Действительно, LinkedHashMap расширяет класс HashMap и реализует интерфейс Map, но что же в нем такого от связанных списков? Давайте будем разбираться.



LinkedHashMap - создание

public class LinkedHashMap<K,V> **extends** HashMap<K,V> **implements** Map<K,V>

Map<Integer, String> linkedHashMap = new LinkedHashMap<Integer, String>();

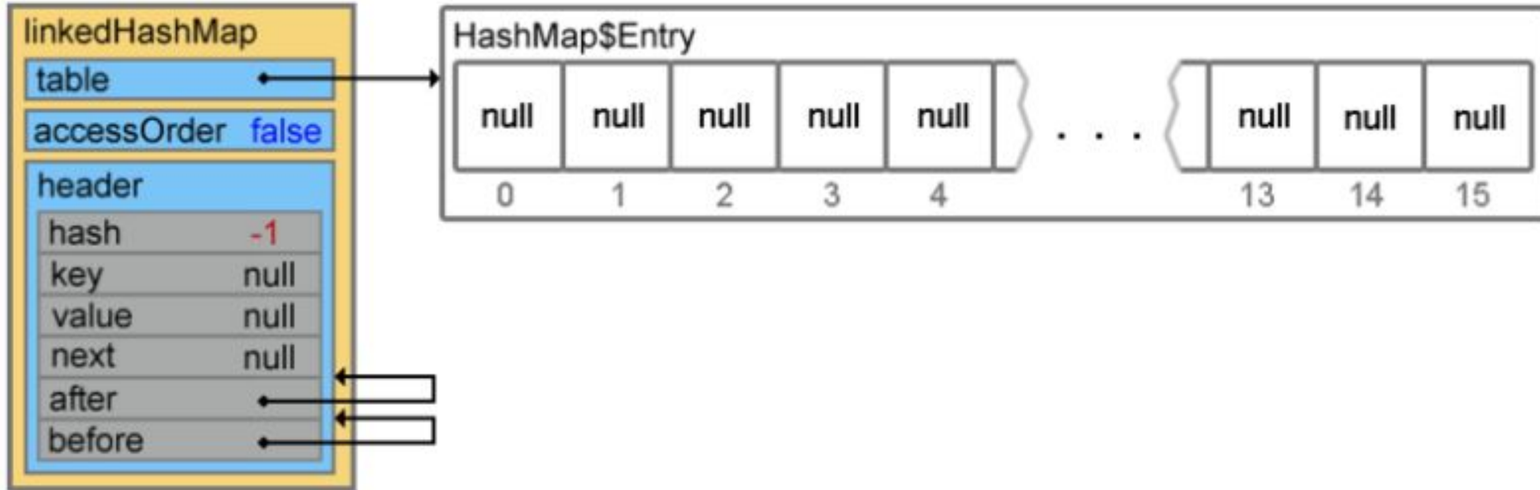
Только что созданный объект linkedHashMap, помимо свойств унаследованных от HashMap (такие как table, loadFactor, threshold, size, entrySet и т.п.), так же содержит два доп. свойства:
header — «голова» двусвязного списка. При инициализации указывает сам на себя;
accessOrder — указывает каким образом будет осуществляться доступ к элементам при использовании итератора. При значении true — по порядку последнего доступа. При значении false доступ осуществляется в том порядке, в каком элементы были вставлены.

LinkedHashMap - создание

Конструкторы класса LinkedHashMap достаточно скучные, вся их работа сводится к вызову конструктора родительского класса и установке значения свойству accessOrder. А вот инициализация свойства header происходит в переопределенном методе init() (теперь становится понятно для чего в конструкторах класса HashMap присутствует вызов этой, ничегонеделающей функции).

```
void init()
{
   header = new Entry<K,V>(-1, null, null, null);
   header.before = header.after = header;
}
```

LinkedHashMap - создание



Новый объект создан, свойства проинициализированы, можно переходить к добавлению элементов.

LinkedHashMap - добавление

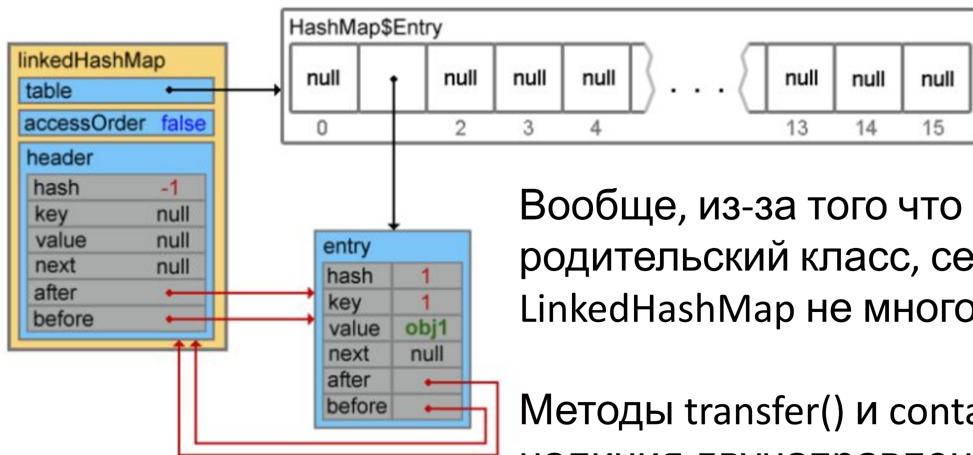
```
linkedHashMap.put(1, "obj1");
```

При добавлении элемента, первым вызывается метод createEntry(hash, key, value, bucketIndex) (по цепочке put() -> addEntry() -> createEntry())

```
void createEntry(int hash, K key, V value, int bucketIndex)
{
    HashMap.Entry<K,V> old = table[bucketIndex];
    Entry<K,V> e = new Entry<K,V>(hash, key, value, old);
    table[bucketIndex] = e;
    e.addBefore(header);
    size++;
}
```

LinkedHashMap - добавление

Первые три строчки добавляют, 4-я делает



Вообще, из-за того что всю основную работу на себя берет родительский класс, серьезных отличий в реализации HashMap и LinkedHashMap не много. Можно упомянуть о парочке мелких:

Методы transfer() и containsValue() устроены чуть проще из-за наличия двунаправленной связи между элементами;
В классе LinkedHashMap.Entry реализованы методы recordRemoval() и recordAccess() (тот самый, который помещает элемент в конец при accessOrder = true). В HashMap оба этих метода пустые.

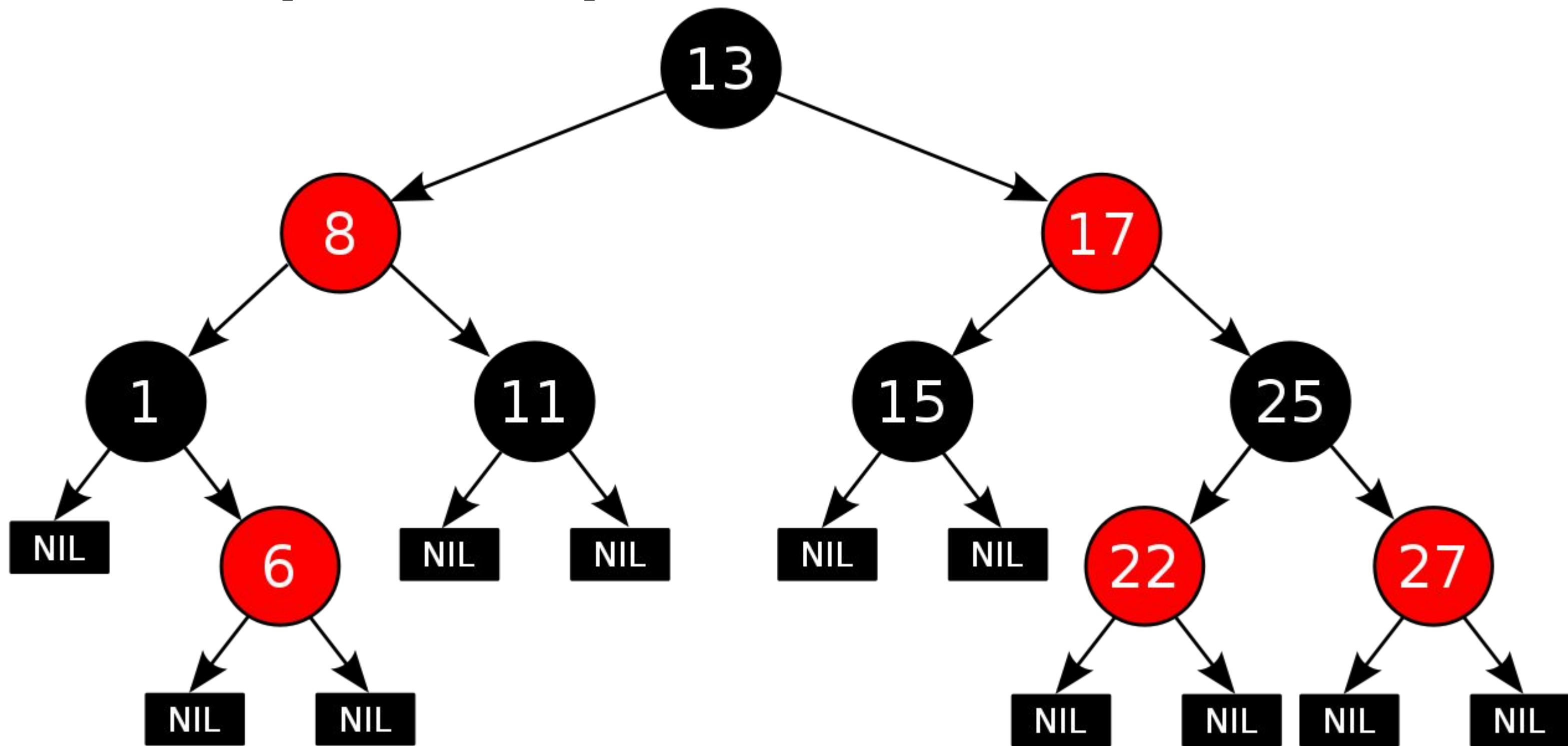
LinkedHashMap - итог

У LinkedHashMap бакеты связаны между собой. Допустим, Вы последовательно добавили в свой мэп значения с ключами 4, 5, 6, 12, 1.

При итерации по ключам HashMap о порядке появления этих ключей судить большого смысла нет, в то время, как LinkedHashMap выдаст 4, 5, 6, 12, 1.

А в зависимости от значения accessOrder поддерживается либо порядок в котором элементы добавляются, либо порядок в котором они извлекаются

Красно-чёрные деревья



Красно-чёрные деревья

Красно-чёрное дерево — двоичное дерево поиска, в котором каждый узел имеет атрибут цвет, принимающий значения красный или чёрный. В дополнение к обычным требованиям, налагаемым на двоичные деревья поиска, к красно-чёрным деревьям применяются следующие требования:

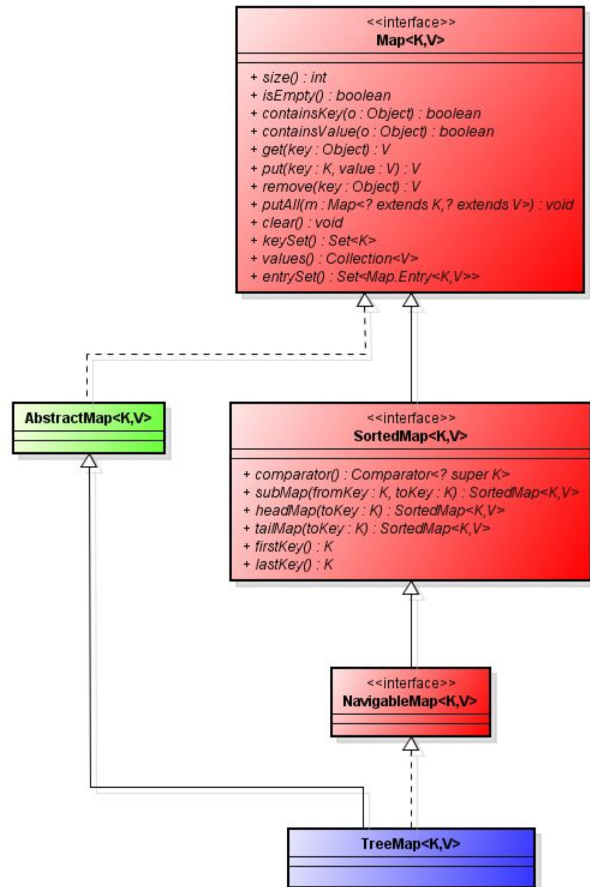
Узел либо красный, либо чёрный.
Корень — чёрный. ☺️
Все листья(NIL) — чёрные.
Оба потомка каждого красного узла — чёрные.
Всякий простой путь от данного узла до любого листового узла, являющегося его потомком, содержит одинаковое число чёрных узлов.

Красно-чёрные деревья

Результатом является то, что дерево примерно сбалансировано. Так как такие операции как вставка, удаление и поиск значений требуют в худшем случае времени, пропорционального длине дерева, эта теоретическая верхняя граница высоты позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

TreeMap

TreeMap основан на Красно-Черном дереве, вследствие чего TreeMap сортирует элементы по ключу в естественном порядке или на основе заданного вами компаратора.TreeMap гарантирует скорость доступа log(n) для операций containsKey, get, put и remove.



TreeMap

Давайте рассмотрим простой пример использования TreeMap

```
Map treeMap = new TreeMap<>();
treeMap.put("Bruce", "Willis");
treeMap.put("Arnold", "Schwarzenegger");
treeMap.put("Jackie", "Chan");
treeMap.put("Sylvester", "Stallone");
treeMap.put("Chuck", "Norris");

for(Map.Entry e : treeMap.entrySet()){

    System.out.println(e.getKey()+" "+ e.getValue());
}
```

TreeMap

Вывод на консоль:

Arnold Schwarzenegger
Bruce Willis
Chuck Norris
Jackie Chan
Sylvester Stallone

Как видим, элементы отсортированы по ключу (Chuck Norris не первый).
Чтобы получить ключи и значения нужно использовать методы keySet() и values().
При попытке добавить null-элемент в TreeMap происходит исключение NullPointerException.

TreeMap - создание

В классе TreeMap присутствуют следующие конструкторы:

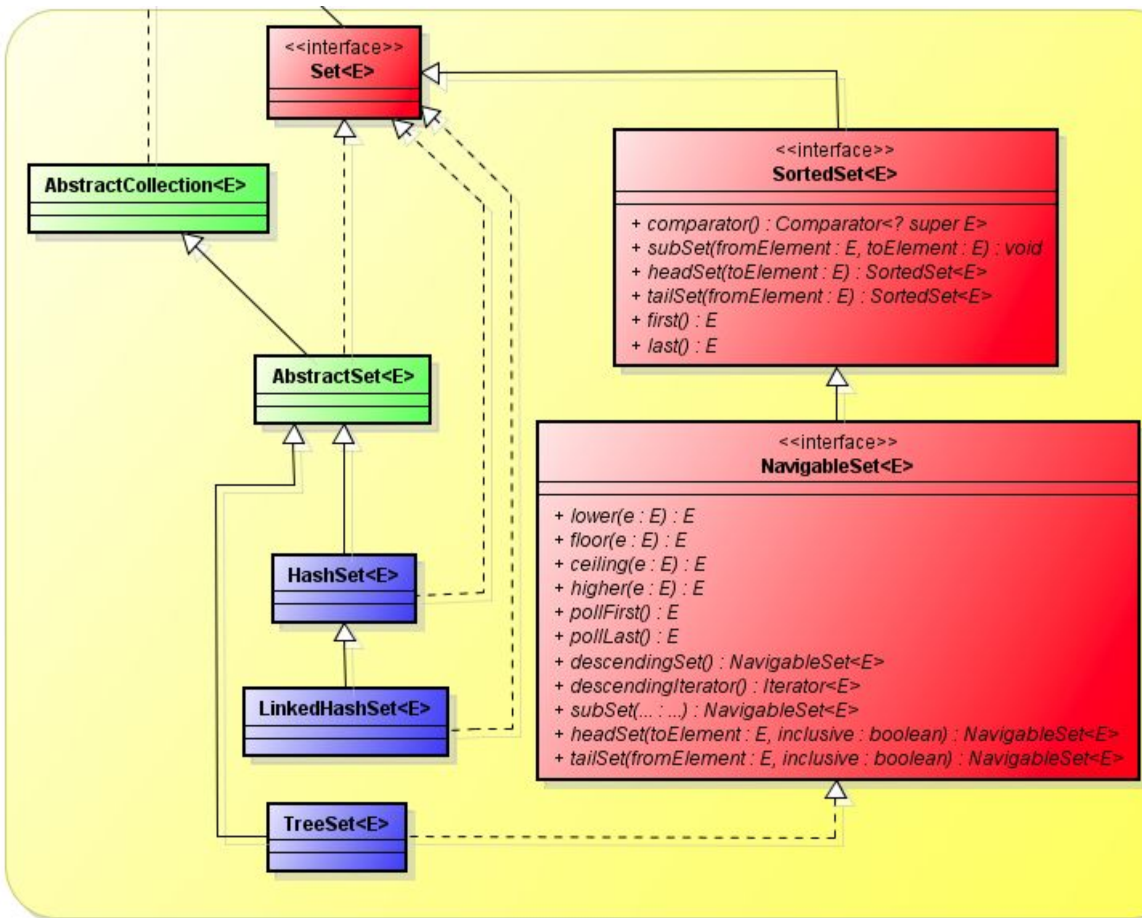
TreeMap( ) , TreeMap(Comparator comp), TreeMap(Map m), TreeMap(SortedMap sm)

Первый конструктор создает коллекцию, в которой все элементы отсортированы в натуральном порядке их ключей.
Второй конструктор создаст пустую коллекцию, элементы в которой будут отсортированы по закону, который определен в передаваемом компараторе.
Третий конструктор создаст TreeMap на основе уже имеющегося Map.
Четвертый конструктор создаст TreeMap на основе уже имеющегося SortedMap , элементы в которой будут отсортированы по закону передаваемой SortedMap .

Обратите внимание на то, что для сортировки используются ключи, а не

Set

Описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества.



HashSet, LinkedHashSet, TreeSet

В основе Map. Изучаем самостоятельно.

Java Generics

Обобщённое программирование — это такой подход к описанию данных и алгоритмов, который позволяет их использовать с различными типами данных без изменения их описания.

```
public class Box {
    private Object object;

    public void set(Object object) {
this.object = object; }
    public Object get() { return object; }
}
```

```
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Java Generics Name Convention

E - Element (used extensively by the Java Collections Framework)
K - Key
N - Number
T - Type
V - Value

Java Generics – множество параметров

package ru.spbstu.generics.multiple;

Java Generics – методы

package ru.spbstu.generics.methods

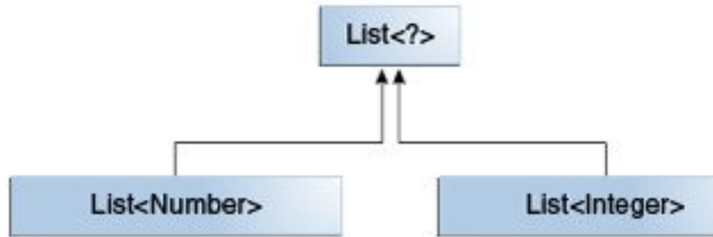
Java Generics – ограничение типизации

package ru.spbstu.generics.bounding

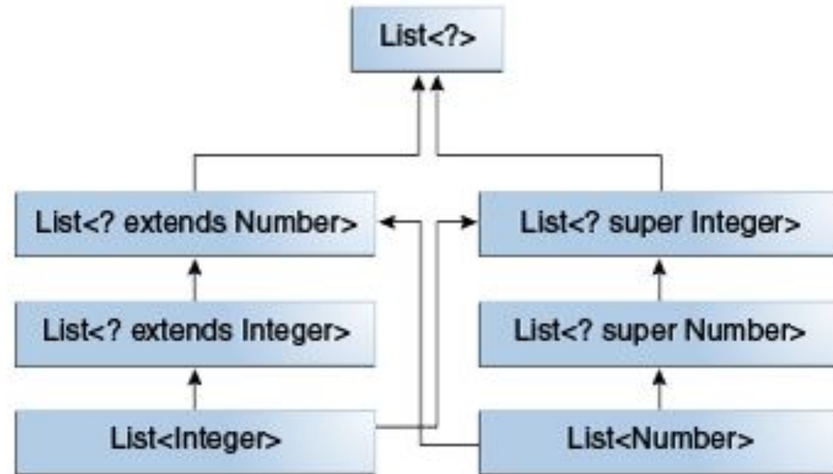
Java Generics – частое заблуждение

Box<Integer> не является подтипом Box<Number> хотя Integer является подтипом Number.

Java Generics – Wildcards



package ru.spbstu.generics.wildcard



Спасибо за внимание!