Коллекции (Collections) в Java. Map.

Цель и задачи лекции

Цель - ознакомиться с Map-коллекциями в Java

Задачи

- Познакомиться с Map
- Познакомиться с HashMap
- Повторить связь Hash и Equals

План занятия

- Map
- HashMap
- Hash & Equals

Вопросы для самоконтроля

- Что такое коллекция Map в Java?
- Что такое HashMap? Приведите пример использования.
- Как работает HashCode и Equals? Когда они равны?

Map

Map не является реализацией интерфейса **Collection**, тем не менее, является частью фреймворка **Collections**.

Map - объект, который хранит пары ключ-значение и не может содержать повторяющихся ключей. При добавлении элемента по существующему ключу происходит запись нового элемента по ключу вместо старого. Это демонстрирует следующий пример:

```
Map<String, String> map = new HashMap<String, String>();
map.put("1", "a");
map.put("1", "2");
for( Entry<String, String> entry : map.entrySet() ){
	System.out.println( entry.getKey() + " " + entry.getValue() );
}
// Output: 1 2
```

Интерфейс **Map** предоставляет три способа для доступа к данным: используя **Set** из ключей (метод **keySet**), коллекцию из значений (метод **values**) и **Set** из пары ключ-значение (метод **entrySet**). Порядок элементов в Map зависит от реализации интерфейса. Интерфейс **Map.Entry**, объект которого возвращает метод **entrySet**(), выглядит следующим образом:

```
public interface Entry {
```

```
        K getKey();
        V getValue();
        V setValue(V value);
}
```

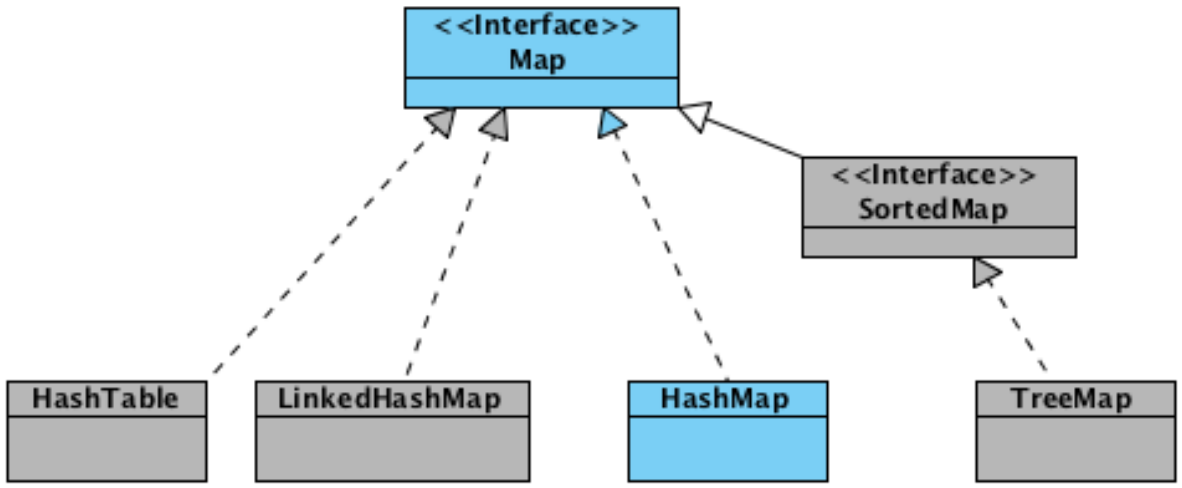
С помощью метода **setValue**() интерфейса **Entry** предоставляется возможность изменять значения во время перебора элементов карты. Например, следующим образом:

```
Map<String, String> map = new HashMap<String, String>();
map.put("1", "a");
map.put("2", "b");
map.put("3", "c");

for( Entry<String, String> entry : map.entrySet() )
        if( "2".equals( entry.getKey() ) )
                entry.setValue( "x" );

System.out.println( map );
// Output: {3=c, 2=x, 1=a}
```

Некоторые карты имеют ограничения на элементы и значения, которые они могут содержать. Например, некоторые реализации запрещают использовать **null** в качестве ключа или значения. Например, при попытке добавить **null-элемент** в **TreeMap** происходит исключение **NullPointerException**. **HashMap** и **LinkedHashMap** позволяют добавлять **null-элементы**. При попытке добавления null-значений в **Map**, который это не поддерживает, необходимо генерировать **NullPointerException** или **ClassCastException**. Второй возможный вариант обработки запрещенных значений - возвращение методом false без генерирования исключения.

Платформа Java версии 1.5 предоставляет три класса общего использования, реализующих интерфейс Map: HashMap, TreeMap и LinkedHashMap.

На Рис.1 приведена иерархия этих классов.

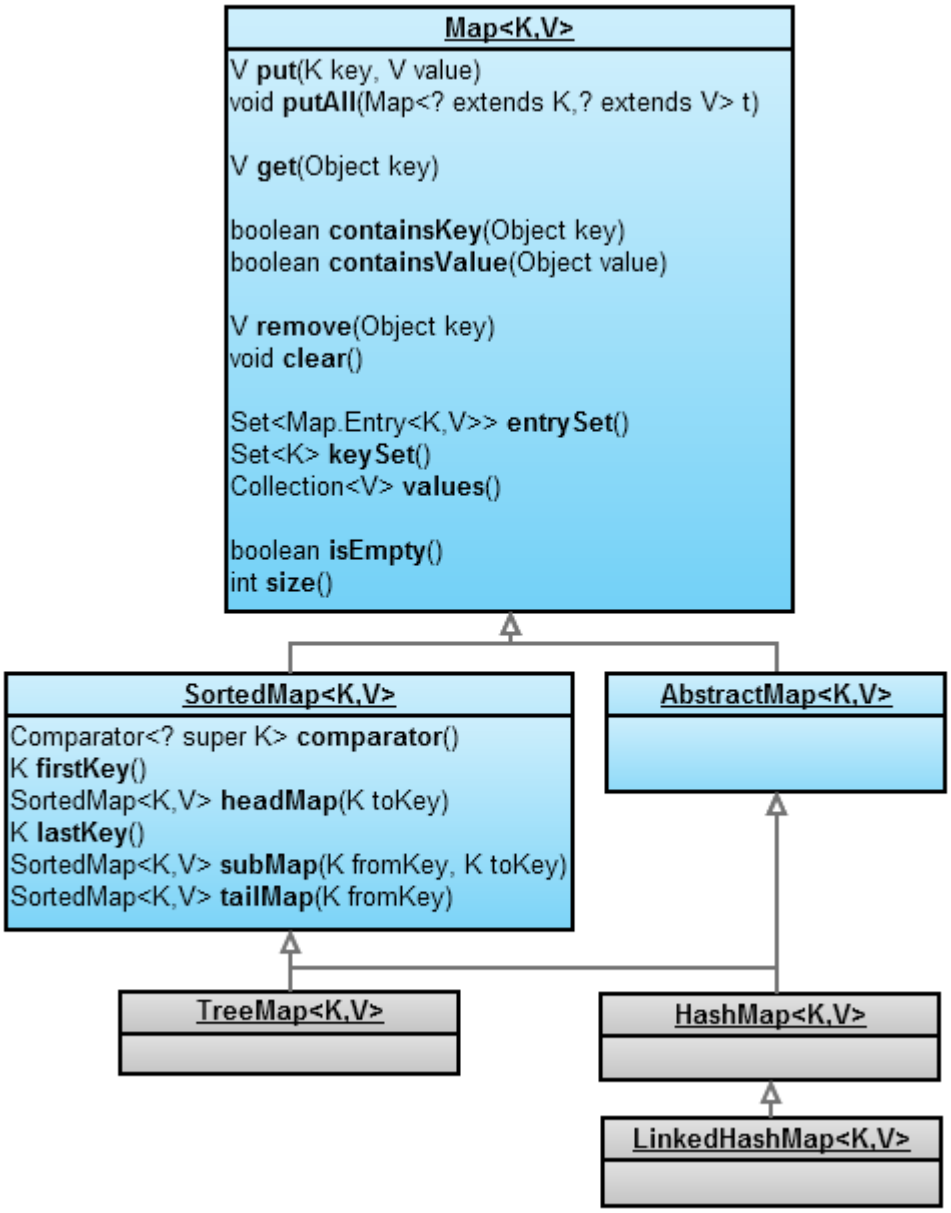


Иерархия HashMap

Интерфейс Map (java.util.Map<K, V>)

1. **Vget(KeyK)** – Возвращает объект, соответствующий указанному ключу или значение null, если карта не содержит указанный ключ. Ключ может быть равен null.
2. **Vput(KeyK, ValueV)** – Добавляет ключ и значение к карте. Если такой ключ уже имеется, то новый объект заменяет предыдущий, связанный с этим ключом. Этот метод возвраща-ет предыдущее значение объекта или значение null, если ключ не содержался в карте ранее. Ключ может быть равен null, но значение должно быть отлично от null.
3. **voidputAll(Map<? extendsK, ? extendsV> entries)** – Добавляет все элементы заданной карты к текущей.
4. **booleancontainsKey(Objectkey)** – Возвращает значение true, если в карте имеется указанный ключ.
5. **booleancontainsValue(Objectvalue)** – Возвращает значение true, если в карте имеется указанное значение.
6. **Set<Map.Entry<K, V>> entrySet()** – Возвращает представление карты в виде множества объектов Map.Entry, т.е. пар "ключ-значение". Из этого представления можно удалять элементы, при этом они удаляются и из карты, но добавлять их нельзя.
7. **Set<K> keySet()** – Возвращает представление карты в виде множества всех ключей. Из этого представления можно удалять элементы, при этом ключи и соответствующие им значения автоматически удаляются из карты, но добавлять новые элементы нельзя.
8. **Collection<V> values()** – Возвращает представление карты в виде множества всех значений. Из этого представления можно удалять элементы, при этом значения и соответствующие им ключи автоматически удаляются из карты, но

добавлять новые элементы нельзя.



Иерархия Map во фреймворке Collections.

Поведение и быстродействие HashMap, TreeMap и LinkedHashMap являются аналогами HashSet, TreeSet и LinkedHashSet соответственно:

- **HashMap** хранит ключи в хеш-таблице, из-за чего имеет наиболее высокую производительность, но не гарантирует порядок элементов. Может содержать как null-ключи, так и null-значения;

- **TreeMap** хранит ключи в отсортированном порядке, из-за чего работает существенно медленнее, чем HashMap. Не может содержать null-ключи, но может

содержать null-значения. Сортироваться элементы будут либо в зависимости от реализации интерфейса Comparable, либо используя объект Comparator, который необходимо передать в конструктор TreeMap;

- **LinkedHashMap** отличается от HashMap тем, что хранит ключи в порядке их вставки в Map. Эта реализаци Map лишь немного медленнее HashMap. Может содержать как null-ключи, так и null-значения.

Интерфейс **Map** содержит устаревшую реализацию **Hashtable**, которую не рекомендуется использовать. Некоторые существенные различия Map и Hashtable приведены ниже:

- Карта обеспечивает перебор элементов с использованием интерфейса Collection вместо прямой поддержки итераций через объект Enumeration. Коллекция помогает значительно повысить выразительность интерфейса.
- Интерфейс Map позволяет производить итерации по ключам, значениям и связке ключ-значение. Hashtable не позволяет производить итерации по связке ключ-значение.
- Map предоставляет безопасный путь удаления элементов в средине итерации, Hashtable - не предоставляет. Например:

```
Map<String, String> map = new HashMap<String, String>();
map.put("1", "a");
map.put("2", "b");
map.put("3", "c");

for (Iterator<String> it = map.keySet().iterator(); it.hasNext(); )
        if ( "2".equals( it.next() ) )
                it.remove();

System.out.println( map );
// Output: {3=c, 1=a}
```

Ниже приведено несколько положений о Map, о которых необходимо помнить:

- Два экземпляра Map равны, если они представляют один и тот же набор пар ключ-значение.
- Особое внимание необходимо уделить ключам, которые могут изменяться. В поведении Map не указано, что делать если значение ключа изменилось таким образом, что ключ стал равен другому ключу в Map.
- Все реализации Map общего назначения должны предоставлять два стандартных конструктора: конструктор по умолчанию, который создает пустую карту и конструктор, который принимает в качестве аргумента Map для создания новой карты с такими же набором ключ-значение. За этим должен следить программист, так как интерфейсы не могут содержать конструкторов.
- Всегда переопределяйте методы equals() и hashCode() так как они могут использоваться в методе contains(), а также для более эффективного расположения элементов в карте, в результате чего поиск происходит быстрее.
- "Деструктивные" методы (то есть, методы, которые могут разрушить целостность карты) должны генерировать исключение UnsupportedOperationException если коллекция не поддерживает операцию. Это рекомендуемое, но не обязательное правило.

Использование Map в Java

Множество представляет собой набор данных, в котором можно быстро найти существующий элемент. Однако для этого нужно иметь точную копию требуемого элемента. Этот вид поиска не очень распространен, поскольку обычно известна лишь некоторая информация (ключ), по которой можно найти соответствующий элемент. Специально для этого предназначена структура данных,поддерживающая отображение, которую называют также картой. Карта хранит пары "ключ-значение". Каждое значение можно найти по его ключу. Например, в таблице могут находиться записи с информацией о сотрудниках, где ключами являются идентификационные номера сотрудников, а значениями — объекты Employee.

Основные реализации Map

В библиотеке Java предусмотрено две основные реализации карт: хэш-карта HashMap и карта-дерево TreeMap. Оба класса реализуют интерфейс Map.
В хэш-карте ключи расположены случайным образом, а в карте-дереве — в строгом порядке. Хэш-функция, или функция сравнения, применяется только для ключей, а са-ми значения, соответствующие этим ключам, не хешируются и не сравниваются.
Какую же из карт следует выбрать? Как и для множеств, хеширование несколько быстрее, поэтому его рекомендуется использовать там, где порядок следования клю-чей не имеет значения.
Ниже показано, как создается хэш-карта для хранения информации о сотрудниках.

```
Map staff = new HashMap< String, Employee>();
//HashMap реализует интерфейс Map
Employee harry = new Employee(“Harry Hacker”);
staff.put(“987-98-9996”, harry);
```

При добавлении объекта к карте должен быть указан и его ключ. В данном случае ключом является строка, а соответствующим значением — объект Employee.
Чтобы обратиться к объекту, нужно воспользоваться ключом.

```
String s = “987-98-9996”;
Employee e = staff.get(s);//читает запись harry
```

Если данных, соответствующих указанному ключу в наборе данных нет, метод get() возвращает значение null. Ключи должны быть уникальными: нельзя сохранить два значения с одинаковым ключом. Если вызвать метод put() дважды с одинаковым ключом, то второе значение просто заменит первое. Кроме того, метод put() возвращает предыдущее значение, хранимое с указанным ключом.

Метод remove() удаляет элемент из карты, а метод size() возвращает число элементов карты. В архитектуре наборов данных карта сама по себе не рассматривается как набор. (В других архитектурах структур данных карта считается набором пар, или значений, индексируемых ключами.) Однако в библиотеке Java предусмотрено использование представления (view) карты, которое реализует интерфейс Collection или один из его дочерних интерфейсов.

Существует три типа представлений: в виде множества ключей, набора значений (который не является множеством) или множества пар "ключ-значение". Ключи и пары "ключ-значение" формируют множество, так как в карте может присутствовать только один уникальный экземпляр объекта-ключа. Перечисленные ниже методы возвращают эти три типа представлений карты.

Set keySet()
Collection values()
Set> entrySet()

(Элементы последнего множества пар "ключ-значение" являются объектами внутреннего класса Map.Entry) Обратите внимание, что множество ключей не является объектом HashSet или TreeSet,но представляет собой объект некоторого другого класса, реализующего интерфейс Set.Интерфейс Set расширяет интерфейс Collection.Следовательно, вы можете использовать метод keySet().
Например, можно перебрать все ключи карты:

```
Set keys = map.keySet();
for (String key: keys) {
//действия с ключом
}
```

Если нужно одновременно просматривать ключи и значения, то можно избежать необходимости поиска значений, перечисляя все записи. Для этого можно использовать следующую заготовку кода:

```
for (Map.Entry entry: staff.entrySet()) {
String key = entry.getKey();
Employee value = entry.getValue();
//действия с ключом и значением
}
```

Специальные реализации Map

Хэш-карты с нестрогим кэшированием

Класс хэш-карт с нестрогим кэшированием WeakHashMap был разработан для ре-шения интересной задачи. Что происходит со значением, ключ которого больше не используется в программе, например из-за того, что исчезла последняя ссылка на этот ключ? В этом случае обратиться к объекту-значению уже нельзя. А так как этот ключ уже не содержится нигде в программе, то нет никакой возможности удалить его пару "ключ-значение" из карты. Но почему его не может удалить система сборки мусора, в обязанности которой как раз и входит удаление неиспользуемых объектов?

К сожалению, все не так просто. Средство сборки мусора в системе управления памятью следит за действующими объектами. Пока объект карты активен,все ячейки карты также активны. Таким образом, об удалении неиспользуемых значений из ак-тивных карт должна позаботиться сама программа. Именно для этого и предназначен класс WeakHashMap, Такая структура данных взаимодействует с системой сборки му-сора для удаления тех пар "ключ-значение", для которых единственной ссылкой на ключ является запись в хэш-таблице. Вот как работает этот механизм. Класс WeakHashMap использует для хранения ключей нестрогие ссылки (weak references). Объект WeakReference содержит ссылку на другой объект, т.е. в данном случае на ключ хэш-таблицы. Обычно, если при сборке мусора выясняется, что на некоторый объект нет ссылок, этот объект удаляется. А если единственная ссылка на объект имеет тип WeakReference, эта нестрогая ссылка помещается в очередь. Периоди-чески происходит проверка на появление новых ссылок в очереди, так как это означает, что данный ключ больше не используется и его объект можно удалить. Таким образом, класс WeakHashMap удаляет соответствующее этому ключу значение.

Связанные хэш-карты

В JDK 1.4 были предложены классы LinkedHashSet и LinkedHashMap, которые запоминают последовательность вставки в набор данных новых пунктов. Таким обра-зом, порядок следования пунктов таблицы уже не выглядит случайным. По мере добавления записей в таблицу они формируют двусвязный список.
Рассмотрим, например, карту:

```
Map staff = new LinkedHashMap();
Staff.put("144-25-5464", new Employee("Amy Lee"));
Staff.put("567-24-2546", new Employee("Harry Hacker"));
Staff.put("157-62-7935", new Employee("Gary Cooper"));
Staff.put("456-62-5527", new Employee("Francesca Cruz"));
```

Итератор staff.ketSet().iterator() перечислит ее ключи в следующем порядке:

```
144-25-5464
567-24-2546
157-62-7935
456-62-5527
```

А итератор staff.values().iterator() перечислит ее значения так:

```
Amy Lee
Harry Hacker
Gary Cooper
Francesca Cruz
```

Связная хэш-карта может запоминать порядок доступа и учитывать его при пере-боре элементов. Каждый раз, когда вы вызываете метод get() или put() запись, которую он затрагивает, удаляется из занимаемой позиции и перемещается в конец связного списка. При этих операциях изменяется структура связного списка, но не ячеек хэш-таблицы. Запись остается в той ячейке, которая соответствует хэш-коду ключа. Для того чтобы создать подобную хэш-карту, нужно использовать следующее выражение:

```
LinkedHashMap<k, v="">(initialCapacity, loadFactor, true)
```

Знать порядок доступа необходимо, например, для создания кэша, работающего по принципу "последнего по времени использования". Например, вам может понадо-биться хранить в памяти наиболее часто используемые записи, а те, с которыми приходится работать редко, извлекать из базы данных. Если вы не находите запись в таб-лице, а таблица уже достаточно заполнена, вы можете удалить с помощью итератора первые несколько элементов. Именно эти элементы используются реже других.

Хэш-карты с индивидуальным хэшированием

В JDK 1.4 добавлен еще один специальный класс IdentityHashMap, выполняю-щий индивидуальное хэширование. Хэш-коды ключей в нем подсчитываются не мето-дом hashCode(),а методом System.identityHashCode().Этот метод вычисляет хэш-код по адресу объекта в памяти. Кроме того, для сравнения объектов класс IdentityHashMap применяет оператор ==, а не метод equals().

Иначе говоря, разные объекты считаются отличающимися друг от друга, даже если их содержимое совпадает. Этот класс полезен для реализации алгоритмов обхода объектов (например, для сериализации), в которых требуется следить даже за теми объектами, которые уже были пройдены итератором.

HashMap

Использование Collections в Map

Немодифицируемые представления

Класс Collections содержит методы, которые создают немодифицируемые представления (unmodifiableview) наборов данных. В этих представлениях реализована проверка существующего набора, выполняемая на этапе работы программы. При попытке модифицировать набор, генерируется исключение и набор данных остается неизменным.
Для получения не модифицируемых Map представлений используются методы:
Collections.unmodifiableMap
Collections.unmodifiableSortedMap

Предположим, например, что вы хотите, чтобы некоторый фрагмент вашего кода просматривал, но не затрагивал содержимое набора данных. Для этого выполните следующие действия:

```
Map<String, Employee> staff = new HashMap<String, Employee>();
…
lookAt(new Collections.unmodifiableMap(staff));
```

Метод **Collections.unmodifiableMap** возвращает экземпляр класса, реализующего интерфейс Map. Метод доступа этого класса извлекает значения из набора staff.Очевидно, что метод **lookAt()** может вызывать все методы, объявленные в интерфейсе Map. Однако все модифицирующие методы переопределены так, что вместо обращения к базовому набору генерируют исключение UnsupportedOperationException.

Не модифицирующее представление не делает сам набор данных неизменяемым. Вы можете модифицировать набор посредством обычной ссылки (в наше случае это staff).При этом методы, модифицирующие элементы набора, остаются доступными.

Синхронизируемые представления

Если вы обращаетесь к набору данных из нескольких потоков, необходимо при¬нять меры, чтобы не повредить информацию в наборе. Это неминуемо произойдет, если, например, один поток будут пытаться включить элемент в хэш-таблицу, а другой — перегенерировать ее. Вместо того чтобы реализовать классы наборов данных, обеспечивающих безопасную работу с потоками, разработчики библиотеки предпочли использовать для этого механизм представлений. Например, статический метод synchronizedMap() класса Collections может преобразовать любую карту в Map с синхронизированными методами доступа.

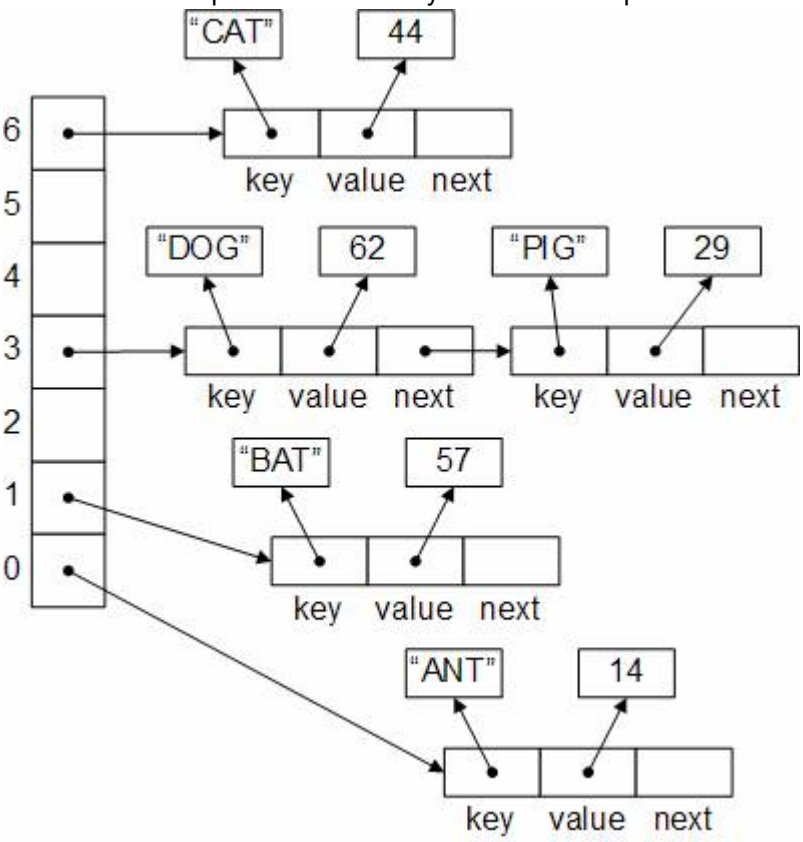
```
HashMap<String, Employee> hashMap = new HashMap<String, Employee>();
Map<String, Employee> map = Collections.synchronizedMap(hashMap);
```

Теперь вы можете обращаться к объекту map из различных потоков. Такие методы, как get() и put(),сериализованы: каждый метод должен полностью закончить свою работу перед тем, как другой поток может вызвать подобный метод.

При разработке программы необходимо следить, чтобы ни один поток не обращался к структуре данных посредством обычных де синхронизированных методов. Самый простой способ обеспечить это — не сохранять ни одной ссылки на базовый объект.

Классы Hashtable и Dictionary

Традиционный класс Hashtable служит той же цели, что и HashMap, и имеет, в сущности, такой же интерфейс. Как и методы класса Vector, методы класс Hashtable синхронизированы. Если Вам не требуется обеспечить синхронизацию или совместимость с кодом для предыдущих версий платформы Java, то в таком случае следует воспользоваться классом HashMap. Класс Dictionary является абстрактным классом-родителем Hashtable.



Пример HasHmap с данными

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. Разрешение коллизий осуществляется с помощью метода цепочек.

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;

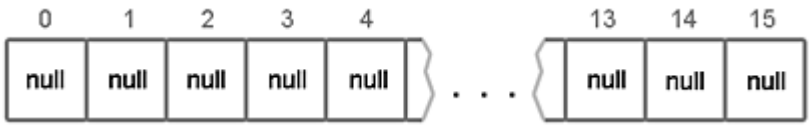
- loadFactor — Коэффициент загрузки. Значение по умолчанию 0.75 является хорошим компромиссом между временем доступа и объемом хранимых данных;

- 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];
```



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).

2. Далее генерируется хэш на основе ключа. Для генерации используется метод hash(hashCode), в который передается key.hashCode().

3.
```
static int hash(int h)
{
 h ^= (h >>> 20) ^ (h >>> 12);
   return h ^ (h >>> 7) ^ (h >>> 4);
```

```
}
```

4. 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

5. С помощью метода indexFor(hash, tableLength), определяется позиция в массиве, куда будет помещен элемент.

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

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

```
h & (length - 1) = 3
```

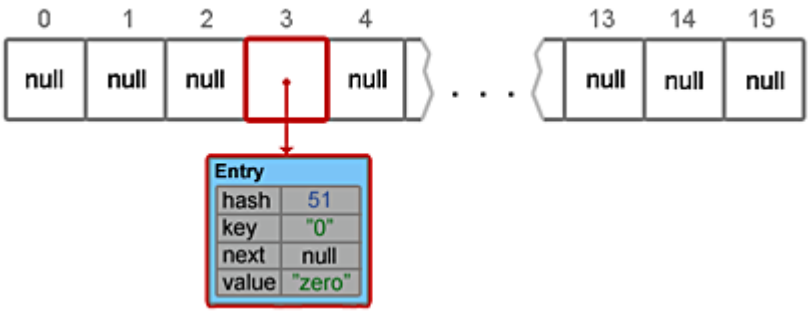
7. Теперь, зная индекс в массиве, мы получаем список (цепочку) элементов, привязанных к этой ячейке. Хэш и ключ нового элемента поочередно сравниваются с хэшами и ключами элементов из списка и, при совпадении этих параметров, значение элемента перезаписывается.

```
if (e.hash == hash && (e.key == key || key.equals(e.key)))
{
    V oldValue = e.value;
    e.value = value;

    return oldValue;
}
```

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

```
void addEntry(int hash, K key, V value, int index)
{
    Entry<K, V> e = table[index];
    table[index] = new Entry<K, V>(hash, key, value, e);
    ...
}
```

9.



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

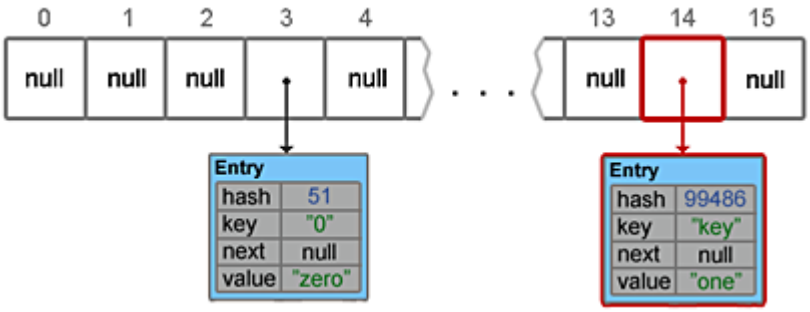
h ^ (h >>> 20) ^ (h >>> 12) = 106054

h ^ (h >>> 7) ^ (h >>> 4) = 99486

3. Определение позиции в массиве

h & (length - 1) = 14

4. Подобные элементы не найдены
5. Добавление элемента



---

hashmap.put(null, null);

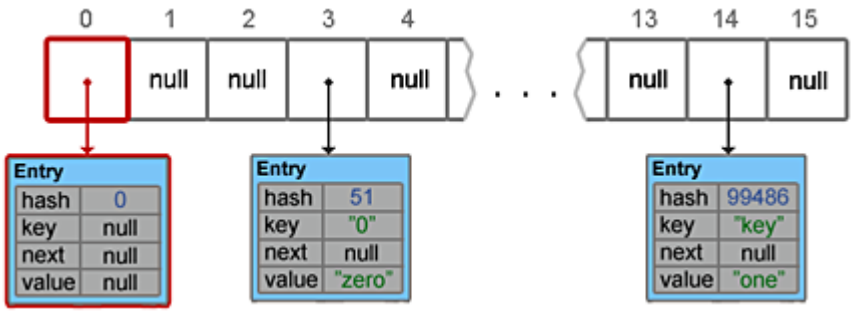
Footprint{Objects=13, References=33, Primitives=[int x 18, char x 11, float]}
Object size: 376 bytes

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().

3. addEntry(0, null, value, 0);
4.



---

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

h ^ (h >>> 20) ^ (h >>> 12) = 104100

h ^ (h >>> 7) ^ (h >>> 4) = 101603

3. Определение позиции в массиве

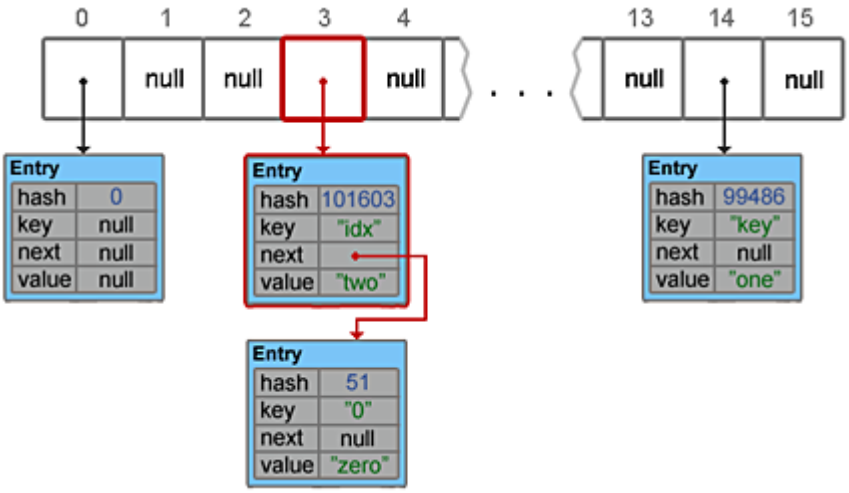
h & (length - 1) = 3

4. Подобные элементы не найдены

5. Добавление элемента

```
// В table[3] уже хранится цепочка состоящая из элемента ["0", "zero"]
Entry<K, V> e = table[index];

// Новый элемент добавляется в начало цепочки
table[index] = new Entry<K, V>(hash, key, value, e);
```

6.



Resize и Transfer

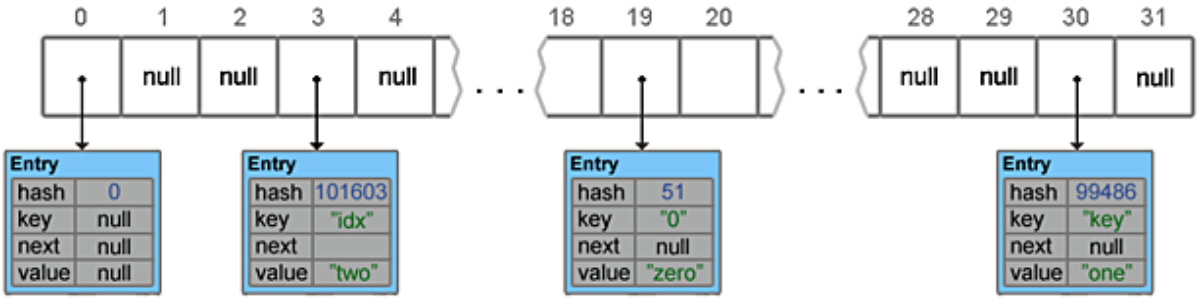
Когда массив table[] заполняется до предельного значения, его размер увеличивается вдвое и происходит перераспределение элементов. Как вы сами можете убедиться, ничего сложного в методах resize(capacity) и transfer(newTable) нет.

```
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, ещё 15 элементов, то в результате размер будет увеличен и распределение элементов изменится.



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.

Hash & Equals

Хэш-код объекта, hashCode (повторение, подробно в 5.4)

В классе *Object*, который является родительским классом для объектов java, определен метод **hashCode()**, позволяющий получить уникальный целый номер для данного объекта. Когда объект сохраняют в коллекции типа HashSet, то данный номер позволяет быстро определить его местонахождение в коллекции и извлечь. Функция **hashCode()** объекта *Object*

возвращает целое число *int*, размер которого равен 4-м байтам и значение которого располагается в диапазоне от -2 147 483 648 до 2 147 483 647.
Рассмотрим простой пример *HashCodeTest.java*, который в консоли будет печатать значение *hashCode*.

```
public class HashCodeTest
{
   public static void main(String[] args)
   {
      int hCode = (new Object()).hashCode();
      System.out.println("hashCode = " + hCode);
   }
}
```

Значение **hashCode** программы можно увидеть в консоли.

```
hashCode = 954599881
```

По умолчанию, функция *hashCode()* для объекта возвращает номер ячейки памяти, где объект сохраняется. Поэтому, если изменение в код приложения не вносятся, то функция должна выдавать одно и то же значение. При незначительном изменении кода значение *hashCode* также изменится.

Функция сравнения объектов equals()

В родительском классе *Object* наряду с функцией *hashCode()* имеется еще и логическая функция *equals(Object)*/ Функция *equals(Object)*используется для проверки равенства двух объектов. Реализация данной функции по умолчанию просто проверяет по ссылкам два объекта на предмет их эквивалентности.
Рассмотрим пример сравнения двух однотипных объектов Test следующего вида :

```
class Test
{
   int f1 = -1;
   int f2 = -1;

   public Test (final int f1, final int f2)
   {
      this.f1 = f1;
      this.f2 = f2;
   }
   public int getF1()
   {
      return this.f1;
   }
   public int getF2()
   {
      return this.f2;
   }
}
```

Создадим 2 объекта типа Test с одинаковыми значениями и сравним объекты с использованием функции **equals()**.

```
public class EqualsExample
{
   public static void main(String[] args)
   {
```

```
        Test obj1 = new Test(11, 12);
        Test obj2 = new Test(11, 12);
        System.out.println("Объекты :\n\tobj1 = " + obj1 +
                           "\n\tobj2 = " + obj2);
        System.out.println("hashCode объектов :" +
                           "\n\tobj1.hashCode = " + obj1.hashCode() +
                           "\n\tobj2.hashCode = " + obj2.hashCode());
        System.out.println("Сравнение объектов :" +
                           "\n\tobj1.equals(obj2) = " + obj1.equals(obj2));
    }
}
```

Результат выполнения программы будет выведен в консоль :

```
Объекты :
    obj1 = example.Test@780324ff
    obj2 = example.Test@16721ee7
hashCode объектов :
    obj1.hashCode = 2013471999
    obj2.hashCode = 376577767
Сравнение объектов :
    obj1.equals(obj2) = false
```

Не трудно было догадаться, что результат сравнения двух объектов вернет «false». На самом деле, это не совсем верно, поскольку объекты идентичны и в *real time application* метод должен вернуть true. Чтобы достигнуть этого корректного поведения, необходимо переопределить метод *equals()* объекта *Test*.

```
@Override
public boolean equals(Object obj)
{
    if (this == obj)
        return true;
    else if (obj == null)
        return false;
    else if (getClass() != obj.getClass())
        return false;

    Test other = (Test) obj;
    if (f1 != other.getF1())
        return false;
    else if (f2 != other.getF2())
        return false;
    return true;
}
```

Вот теперь функция сравнения *equals()* возвращает значение «true». Достаточно ли этого? Попробуем добавить объекты в коллекцию *HashSet*и посмотрим, сколько объектов будет в коллекции? Для этого в в метод *main* примера *EqualsExample* добавим следующие строки :

```
Set<Test> objects = new HashSet<Test>();
objects.add(obj1);
objects.add(obj2);

System.out.println("Коллекция :\n\t" + objects);
```

Однако в коллекции у нас два объекта.

```
Коллекция :
```

```
        [example.Test@780324ff, example.Test@16721ee7]
```

Поскольку Set содержит только уникальные объекты, то внутри HashSet должен быть только один экземпляр. Чтобы этого достичь, объекты должны возвращать одинаковый **hashCode**. То есть нам необходимо переопределить также функцию **hashCode()** вместе с *equals()*.

```
@Override
public int hashCode()
{
   final int prime = 31;
   int result = 1;
   result = prime * result + f1;
   result = prime * result + f2;
   return result;
}
```

Вот теперь все будет корректно выполняться - для двух объектов с одинаковыми параметрами функция *equals()* вернет значение «true», и в коллекцию попадет только один объект. В консоль будет выведен следующий текст работы программы :

```
Объекты :
        obj1 = example.Test@696
        obj2 = example.Test@696
hashCode объектов :
        obj1.hashCode = 1686
        obj2.hashCode = 1686
Сравнение объектов :
        obj1.equals(obj2) = true
Коллекция :
        [example.Test@696]
```

Таким образом, переопределяя методы *hashCode()* и *equals()* мы можем корректно управлять нашими объектами, не допуская их дублирования.

Правила переопределения equals

Метод equals() необходим в Java для подтверждения или отрицания того факта, что два объекта одного происхождения являются *логически равными*. То есть, сравнивая два объекта, программисту необходимо понять, эквивалентны ли их *значимые поля*. Не обязательно все поля должны быть идентичны, так как метод equals() подразумевает именно *логическое равенство*.

Но иногда нет особой необходимости в использовании этого метода. Как говорится, самый легкий путь избежать проблем, используя тот или иной механизм — не использовать его. Также следует заметить, что однажды нарушив контракт equals вы теряете контроль над пониманием того, как другие объекты и структуры будут взаимодействовать с вашим объектом. И впоследствии найти причину ошибки будет весьма затруднительно.

Когда не стоит переопределять этот метод

- Когда каждый экземпляр класса является уникальным.
- В большей степени это касается тех классов, которые предоставляют определенное поведение, нежели предназначены для работы с данными. Таких, например, как класс Thread. Для них реализации метода equals, предоставляемого классом Object, более

чем достаточно. Другой пример — классы перечислений (Enum).Когда на самом деле от класса не требуется определять эквивалентность его экземпляров.

- Например для класса java.util.Random вообще нет необходимости сравнивать между собой экземпляры класса, определяя, могут ли они вернуть одинаковую последовательность случайных чисел. Просто потому, что природа этого класса даже не подразумевает такое поведение.Когда класс, который вы расширяете, уже имеет свою реализацию метода equals и поведение этой реализации вас устраивает.
- Например, для классов Set, List, Map реализация equals находится в AbstractSet, AbstractList и AbstractMap соответственно.И, наконец, нет необходимости перекрывать equals, когда область видимости вашего класса является private или package-private и вы уверены, что этот метод никогда не будет вызван.

Контракт equals

При переопределении метода equals разработчик должен придерживаться основных правил, определенных в спецификации языка Java:

- Рефлексивность
  для любого заданного значения x, выражение x.equals(x) должно возвращать true.
  Заданного — имеется в виду такого, что x != null
- Симметричность
  для любых заданных значений x и y, x.equals(y) должно возвращать true только в том случае, когда y.equals(x)возвращает true.
- Транзитивность
  для любых заданных значений x, y и z, если x.equals(y) возвращает true и y.equals(z) возвращает true, x.equals(z) должно вернуть значение true.
- Согласованность
  для любых заданных значений x и y повторный вызов x.equals(y) будет возвращать значение предыдущего вызова этого метода при условии, что поля, используемые для сравнения этих двух объектов, не изменялись между вызовами.
- Сравнение null
  для любого заданного значения x вызов x.equals(null) должен возвращать false.

Нарушение контракта equals

Многие классы, например классы из Java Collections Framework, зависят от реализации метода equals(), поэтому не стоит им пренебрегать, т.к. нарушение контракта этого метода может привести к нерациональной работе приложения и в таком случае найти причину будет достаточно трудно.

Согласно принципу рефлексивности, каждый объект должен быть эквивалентен самому себе. Если этот принцип будет нарушен, при добавлении объекта в коллекцию и при последующем поиске его с помощью метода contains() мы не сможем найти тот объект, который только что положили в коллекцию.

Условие симметричности гласит, что два любых объекта должны быть равны независимо от того, в каком порядке они будут сравниваться. Например, имея класс, содержащий всего одно поле строкового типа, будет неправильно сравнивать в методе equals данное поле со строкой. Т.к. в случае обратного сравнения метод всегда вернет значение false.

```
// Нарушение симметричности
```

```
public class SomeStringify {
    private String s;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o instanceof SomeStringify) {
            return s.equals(((SomeStringify) o).s);
        }
        // нарушение симметричности, классы разного происхождения
        if (o instanceof String) {
            return s.equals(o);
        }
        return false;
    }
}


//Правильное определение метода equals
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    return o instanceof SomeStringify &&
            ((SomeStringify) o).s.equals(s);
}
```

Из условия транзитивности следует, что если любые два из трех объектов равны, то в таком случае должны быть равны все три. Этот принцип легко нарушить в том случае, когда необходимо расширить некий базовый класс, добавив к нему *значимый компонент*.

Например, к классу Point с координатами x и y необходимо добавить цвет точки, расширив его. Для этого потребуется объявить класс ColorPoint с соответствующим полем color. Таким образом, если в расширенном классе вызывать метод equals родителя, а в родительском будем считать, что сравниваются только координаты x и y, тогда две точки разного цвета, но с одинаковыми координатами будут считаться равными, что неправильно.

В таком случае, необходимо научить производный класс различать цвета. Для этого можно воспользоваться двумя способами. Но один будет нарушать правило *симметричности*, а второй — *транзитивности*.

```
// Первый способ, нарушая симметричность
// Метод переопределен в классе ColorPoint
@Override
public boolean equals(Object o) {
    if (!(o instanceof ColorPoint)) return false;
    return super.equals(o) && ((ColorPoint) o).color == color;
}
```

В этом случае вызов point.equals(colorPoint) вернет значение true, а сравнение colorPoint.equals(point) —false, т.к. ожидает объект “своего” класса. Таким образом и нарушается правило симметричности.

Второй способ подразумевает делать “слепую” проверку, в случае, когда нет данных о цвете точки, т. е. имеем класс Point. Или же проверять цвет, если информация о нем доступна, т. е. сравнивать объект класса ColorPoint.

```
// Метод переопределен в классе ColorPoint
@Override
public boolean equals(Object o) {
    if (!(o instanceof Point)) return false;

    // Слепая проверка
    if (!(o instanceof ColorPoint))
        return super.equals(o);

    // Полная проверка, включая цвет точки
    return super.equals(o) && ((ColorPoint) o).color == color;
}
```

Принцип *транзитивности* здесь нарушается следующим образом. Допустим, есть определение следующих объектов:

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

Таким образом, хоть и выполняется равенство p1.equals(p2) и p2.equals(p3), p1.equals(p3) вернет значение false. При этом второй способ, на мой взгляд, выглядит менее привлекательным, т.к. в некоторых случаях алгоритм может ослепнуть и не выполнить сравнение в полной мере, а вы об этом можете и не узнать.

Правила переопределения hashCode

Хэш — это некоторое число, генерируемое на основе объекта и описывающее его состояние в какой-то момент времени. Это число используется в Java преимущественно в хэш-таблицах, таких как HashMap. При этом хэш-функция получения числа на основе объекта должна быть реализована таким образом, чтобы обеспечить относительно равномерное распределение элементов по хэш-таблице. А также минимизировать вероятность появления коллизий, когда по разным ключам функция вернет одинаковое значение. (**Подробнее в 5.4**)

Контракт hashCode

Для реализации хэш-функции в спецификации языка определены следующие правила:

- вызов метода hashCode один и более раз над одним и тем же объектом должен возвращать одно и то же хэш-значение, при условии что поля объекта, участвующие в вычислении значения, не изменились.
- вызов метода hashCode над двумя объектами должен всегда возвращать одно и то же число, если эти объекты равны (вызов метода equals для этих объектов возвращает true).
- вызов метода hashCode над двумя неравными между собой объектами должен возвращать разные хэш-значения. Хотя это требование и не является обязательным,

следует учитывать, что его выполнение положительно повлияет на производительность работы хэш-таблиц.

-

Методы equals и hashCode необходимо переопределять вместе

Исходя из описанных выше контрактов следует, что переопределяя в своем коде метод equals, необходимо всегда переопределять и метод hashCode. Так как фактически два экземпляра класса отличаются, потому что находятся в разных областях памяти, сравнивать их приходится по некоторым логическим признакам. Соответственно, два логически эквивалентных объекта, должны возвращать одинаковое значение хэш-функции.

Что произойдет, если будет переопределен только один из этих методов?

1. equals есть, hashCode нет
   Допустим мы правильно определили метод equals в нашем классе, а метод hashCode решили оставить как он есть в классе Object. Тогда с точки зрения метода equals два объекта будут логически равны, в то время как с точки зрения метода hashCode они не будут иметь ничего общего. И, таким образом, помещая некий объект в хэш-таблицу, мы рискуем не получить его обратно по ключу.
   Например, так:

```
Map<Point, String> m = new HashMap<>();
m.put(new Point(1, 1), “Point A”);
// pointName == null
```

2. String pointName = m.get(new Point(1, 1));
3. Очевидно, что помещаемый и искомый объект — это два разных объекта, хотя они и являются логически равными. Но, т.к. они имеют разное хэш-значение, потому что мы нарушили контракт, можно сказать, что мы потеряли свой объект где-то в недрах хэш-таблицы.
4. hashCode есть, equals нет.
   Что будет если мы переопределим метод hashCode, а реализацию метода equals наследуем из класса Object. Как известно метод equals по умолчанию просто сравнивает указатели на объекты, определяя, ссылаются ли они на один и тот же объект. Предположим, что метод hashCode мы написали по всем канонам, а именно — сгенерировали средствами IDE, и он будет возвращать одинаковые хэш-значения для логически одинаковых объектов. Очевидно, что тем самым мы уже определили некоторый механизм сравнения двух объектов.

   Следовательно, пример из предыдущего пункта по идее должен выполняться. Но мы по-прежнему не сможем найти наш объект в хэш-таблице. Хотя будем уже близки к этому, потому что как минимум найдем корзину хэш-таблицы, в которой объект будет лежать.

   Для успешного поиска объекта в хэш-таблице помимо сравнения хэш-значений ключа используется также определение логического равенства ключа с искомым объектом. Т. е. без переопределения метода equals никак не получится обойтись.

Вместо заключения

Таким образом, мы видим, что методы equals и hashCode играют четко определенную роль в языке Java и предназначены для получения характеристики логического равенства двух объектов. В случае с методом equals это имеет прямое отношение к сравнению объектов, в случае с hashCode косвенное, когда необходимо, скажем так, определить примерное

расположение объекта в хэш-таблицах или подобных структурах данных с целью увеличения скорости поиска объекта.

Помимо контрактов equals и hashCode имеется еще одно требование, относящееся к сравнению объектов. Это согласованность метода compareTo интерфейса Comparable с методом equals. Данное требование обязывает разработчика всегда возвращать x.equals(y) == true, когда x.compareTo(y) == 0. Т. е. мы видим, что логическое сравнение двух объектов не должно противоречить нигде в приложении и всегда быть согласованным.

Литература и ссылки

- https://javarush.ru/groups/posts/1340-peregruzka-metodov-equals-i-hashcode-v-java
- https://javarush.ru/groups/posts/1989-kontraktih-equals-i-hashcode-ili-kak-ono-vsje-tam
- https://habr.com/ru/post/168195/