**Работа с Map**

Напомним, что **Map** — это структурированные данные, состоящие из набора пар ключ-значение, и каждый ключ может использоваться только один раз в одной Map. Эта тема раскрывает 9 основных вопросов об использовании [Map](https://docs.oracle.com/javase/7/docs/api/java/util/Map.html) в Java и её воплощённых (имплементированных) классах. Для простоты, я буду использовать в примерах [обобщения](https://docs.oracle.com/javase/tutorial/java/generics/). Потому, я буду писать просто Map, не конкретизируя спецификатор Map. Но вы можете полагать, что оба значения **K** и **V** сопоставимы, что означает **K** расширяет **Comparable** и **V** так же расширяет **Comparable**.

**0. Обращение Map в List**

В Java, интерфейс [Map](https://docs.oracle.com/javase/7/docs/api/java/util/Map.html) предлагает три вида коллекций: набор ключей, набор значений и набор ключ-значение. Все они могут быть обращены в [List](https://docs.oracle.com/javase/7/docs/api/java/util/List.html) при помощи конструктора или метода addAll(). Следующая вырезка кода демонстрирует как сделать [ArrayList](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html) из Map.

//лист ключей

List keyList = **new** ArrayList(Map.keySet());

//лист значений

List valueList = **new** ArrayList(Map.valueSet());

//лист ключ-значения

List entryList = **new** ArrayList(Map.entrySet());

**1. Пройтись по всем значениям в Map**

Проход по каждой паре ключ-значение — самая базовая, основная процедура прохода по Map. В Java, каждая пара хранится в поле Map называемом [Map.Entry](https://docs.oracle.com/javase/7/docs/api/java/util/Map.Entry.html). Map.entrySet() возвращает набор ключ-значений, потому самым эффективным способом пройтись по всем значениям Map будет:

**for**(Map.Entry<--,--> entry: Map.entrySet()) {

//получить ключ

K key = entry.getKey();

//получить значение

V value = entry.getValue();

}

Так же мы можем использовать Iterator, особенно в версиях младше JDK 1.5

Iterator itr = Map.entrySet().iterator();

**while**(itr.hasNext()) {

Entry entry = itr.next();

//получить ключ

K key = entry.getKey();

//получить значение

V value = entry.getValue();

}

**2. Упорядочивание Map по ключам**

Упорядочивание Map по ключам ещё одна часто встречаемая процедура. Первый способ: добавить [Map.Entry](https://docs.oracle.com/javase/7/docs/api/java/util/Map.Entry.html) в список, и упорядочить с использованием компаратора, что сортирует по значениям.

List list = **new** ArrayList(Map.entrySet());

Collections.sort(list, **new** Comparator() {

@Override

**public** **int** compare(Entry e1, Entry e2) {

**return** e1.getKey().compareTo(e2.getKey());

}

});

Другой способ: использовать [SortedMap](https://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html), которая ко всему, ещё и выстраивает свои ключи по порядку. Но, все ключи при этом должны воплощать [Comparable](https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) или приниматься компаратором. Один из имплементированных классов SortedMap — [TreeMap](https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html). Её конструктор принимает компаратор. Следующий код показывает как превратить обычную Map в упорядоченную.

SortedMap sortedMap = **new** TreeMap(**new** Comparator() {

@Override

**public** **int** compare(K k1, K k2) {

**return** k1.compareTo(k2);

}

});

sortedMap.putAll(Map);

**3. Упорядочивание Map по значениям**

Добавление Map в список и последующая сортировка работают и в данном случае, но нужно в этот раз брать Entry.getValue(). Код ниже почти такой же как и раньше.

List list = **new** ArrayList(Map.entrySet());

Collections.sort(list, **new** Comparator() {

@Override

**public** **int** compare(Entry e1, Entry e2) {

**return** e1.getValue().compareTo(e2.getValue());

}

});

Мы всё ещё можем использовать SortedMap в данном случае, но только если значения уникальны. В таком случае, вы можете обратить пару ключ-значение в значение-ключ. Это решение обладает строгим ограничением, и не рекомендуется мною.

**4. Инициализация статической/неизменной Map**

Когда вы желаете, что бы Map оставалась неизменной, хорошим способом будет скопировать оную в неизменяемую (immutable) Map. Такая защитная техника программирования поможет вам создать не только безопасную для использования, но и так же потокобезопасную Map. Для инициализации статической/неизменной Map, мы можем использовать инициализатор static (см. ниже). Проблема данного кода в том, что не смотря на объявление Map как static final, мы всё ещё можем работать с ней после инициализации, например Test.Map.put(3,"three");. Так что это не настоящая неизменность. Для создания неизменяемой Map с использованием статического инициализатора, нам нужен супер анонимный класс, который мы добавим в неизменяемую Map на последнем шаге инициализации. Пожалуйста, посмотрите на вторую часть кода. Когда будет выброшено [UnsupportedOperationException](https://docs.oracle.com/javase/7/docs/api/java/lang/UnsupportedOperationException.html), если вы запустите Test.Map.put(3,"three");.

**public** **class** Test {

**private** **static** **final** Map Map;

**static** {

Map = **new** HashMap();

Map.put(1, "one");

Map.put(2, "two");

}

}

**public** **class** Test {

**private** **static** **final** Map Map;

**static** {

Map aMap = **new** HashMap();

aMap.put(1, "one");

aMap.put(2, "two");

Map = Collections.unmodifiableMap(aMap);

}

}

Библиотека [Guava](https://github.com/google/guava) так же поддерживает различные способы инициализации статических и неизменных коллекций. Чтобы изучить подробнее преимущества утилиты Guava для неизменных коллекций, обратитесь к разделу [Неизменные коллекции в Инструкции Guava](https://github.com/google/guava).

**5. Разница между HashMap, TreeMap, и Hashtable**

Есть три основных воплощения интерфейса [Map](https://docs.oracle.com/javase/7/docs/api/java/util/Map.html) в Java: [HashMap](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html), [TreeMap](https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html), и [Hashtable](https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html). Главные отличия заключаются в следующем:

* **Порядок прохода**. [HashMap](https://javarush.ru/groups/posts/732-kak-rabotaet-hashmap-v-java" \t "_blank) и HashTable не дают гарантий по упорядоченности в Map; в частности, они не гарантируют что порядок останется тем же самым в течении времени. Но TreeMap будет упорядочивать все значения в "естественном порядке" ключей или по компаратору.
* **Допустимые пары ключ-значение.** HashMap позволяет иметь ключ null и значение null. HashTable не позволяет ключ null или значение null. Если TreeMap использует естественный порядок или компаратор не позволяет использовать ключ null, будет выброшено исключение.
* **Синхронизация**. Только HashTable синхронизирована, остальные — нет. Но, "если потокобезопасное воплощение не нужно, рекомендуется использовать HashMap вместо HashTable".

Более подробное сравнение

. | HashMap | HashTable | TreeMap

-------------------------------------------------------

Упорядочивание |нет |нет | да

null в ключ-значение | да-да | нет-нет | нет-да

синхронизировано | нет | да | нет

производительность | O(1) | O(1) | O(log n)

воплощение | корзины | корзины | красно-чёрное дерево

Прочтите подробнее об отношениях [HashMap vs. TreeMap vs. Hashtable vs. LinkedHashMap](https://www.programcreek.com/2013/03/hashmap-vs-treemap-vs-hashtable-vs-linkedhashmap/).

**6. Map с реверсивным поиском/просмотром**

Иногда, нам нужен набор пар ключ-ключ, что подразумевает значения так же уникальны, как и ключи (паттерн один-к-одному). Такое постоянство позволяет создать "инвертированный просмотр/поиск" по Map. То есть, мы можем найти ключ по его значению. Такая структура данных называется [двунаправленная Map](https://en.wikipedia.org/wiki/Bidirectional_map), которая к сожалению, не поддерживается JDK. Обе Apache Common Collections и Guava предлагают воплощение двунаправленной Map, называемые BidiMap и BiMap, соответственно. Обе вводят ограничение, которое задаёт соответствие 1:1 между ключами и значениями.

**7. Поверхностная копия Map**

Почти все, если не все, Map в Java содержат конструктор копирования другой Map. Но процедура копирования не синхронизирована. Что означает когда один поток копирует Map, другой может изменить её структуру. Для предотвращения внезапной рассинхронизации копирования, один из них должен использовать в таком случае Collections.synchronizedMap().

Map copiedMap = Collections.synchronizedMap(Map);

Другой интересный способ поверхностного копирования — использование метода clone(). Но он НЕ рекомендуется даже создателем фреймворка коллекций Java, Джоном Блохом. В споре "[Конструктор копирования против клонирования](https://www.artima.com/intv/bloch13.html)", он занимает позицию: Цитата: *"Я часто привожу публичный метод clone в конкретных классах, поскольку люди ожидают их там увидеть. ... это позор, что Клонирование сломано, но это случилось. ... Клонирование это слабое место, и я думаю люди должны быть предупреждены о его ограничениях."* По этой причине, я даже не показываю вам, как использовать метод clone() для копирования Map

**8. Создание пустой Map**

Если Map неизменна, используйте:

Map = Collections.emptyMap();

Или, используйте любое другое воплощение. Например:

Map = **new** HashMap();