HASH HOMEWORK

Name: Nguyen Dang Loc Student ID: SE160199 Class: SE1616

1. Why to use java.util.WeakHashMap map which is so inconsistent and unpredictable in behaviour?

Answer:

In Java Collection Libraries, besides HashMap there is also WeakHashMap almost the same as HashMap, except that WeakHashMap uses weak reference objects. When a key is discarded (not containing any references), its entry is automatically removed from the map, i.e. garbage collector will handle the entry.

When it comes to the huge applications which contain lots of objects, they may run out of memory at any time. We want to free memory by discarding less used <key, value> pair. WeakHashMap, despite its inconsistent and unpredictable behavior, will help to solve this problem. I think it is better to remove less used/unused keys than to run out of memory.

Code

```
WeakHashMap<String, Integer> numbers = new WeakHashMap<>();
String a = new String("A");
Integer va = 10;
String b = new String("B");
Integer vb= 11;

// Inserting elements
numbers.put(a, va);
numbers.put(b, vb);
System.out.println("WeakHashMap: " + numbers);

a= null;
System.gc();
System.out.println("WeakHashMap after garbage collection: " + numbers);
```

Output

```
WeakHashMap: {A=10, B=11}
WeakHashMap after garbage collection: {A=10}
```

2. Let’s say you have to build dictionary and multiple users can add data in that dictionary? And you can use 2 Collection classes? Which Collection classes you will prefer and WHY?

Answer:

TreeMap - dictionary entry container, because:

- Cannot contain duplicate keys (unique words).
- Cannot contain the null key
- Sorted according to the natural ordering of its keys (maintain word list in alphabet order)
- Can use `Collections.synchronizedMap()` to create a synchronized wrapper for multithreading update operations.

TreeSet - meaning of words container

- Cannot contain duplicate meaning.
- Sorted according to the natural order (maintain meaning list of a word in alphabet order)
- Cannot contain the null value

Declaration:

```
TreeMap<String, TreeSet<String>> dictionary = new TreeMap<>();
```

3. How ConcurrentHashMap works? Can 2 threads on the same ConcurrentHashMap object access it concurrently?

Answer:

(1) The **ConcurrentHashMap** class, which implements ConcurrentMap and Serializable interface, is an enhancement of HashMap. ConcurrentHashMap is divided into different '**segments'** based on concurrency level. So multiple threads can access different segments concurrently.

```
// constructor
public ConcurrentHashMap (int initialCapacity, float loadFactor, int concurrencyLevel)

// Initial Capacity: number of elements initially provided.
// (Capacity 10 - can store 10 entries)
// Load-Factor: threshold used to control resizing.
// Concurrency-Level: number of threads concurrently updating the map
```

ConcurrentHashMap's default concurrency level is 16. (i.e., 16 update operations can be executed at the same time by different threads, each working on a distinct bucket).

At a time, threads can perform retrieval operations but for updated in the object, the thread must lock the particular segment in which the thread wants to operate. This locking mechanism is **Segment locking or bucket locking**. Unlike HashTable, retrieval and update operation are happened concurrently without locking the entire map in ConcurrentHashMap.

Hence, ConcurrentHashMap is a good choice because of its performance.

(2) Yes.

When thread locks one segment for updating (add, remove) it does not block it for retrieval (get), allowing another thread to read it.

ConcurrentHashMap allows different threads to read and write in parallel. However, in the worst-case situation, if two objects are located in the same segment, parallel writing is not allowed.

4. How can you sort given HashMap on basis of keys?

Answer:

Using ArrayList

Create a list of keys using **ArrayList** constructor. Then sort the list using **Collections.sort()**.

```
static Map<String, Integer> map = new HashMap<>();
ArrayList<String> sortedKeys = new ArrayList<String>(map.keySet());

Collections.sort(sortedKeys);
```

Using TreeMap

Keys stored in TreeMap are sorted, we can store all of HashMap's entries into the TreeMap by using its constructor or putAll() method.

```
static Map<String, Integer> map = new HashMap<>();

// TreeMap to store values of HashMap
TreeMap<String, Integer> sorted = new TreeMap<>(map);

// Copy all data from hashMap into TreeMap
TreeMap<String, Integer> sorted = new TreeMap<>();
sorted.putAll(map);
```

5. What are different ways of iterating over keys, values and entry in Map?

Answer:

1. Using **iterator** and **Map.Entry**

```
Iterator<Map.Entry<Integer, Integer>> it = map.entrySet().iterator();
while (it.hasNext())
    Map.Entry<Integer, Integer> pair = it.next();
```

2. Using **foreach** and **Map.Entry**

```
for (Map.Entry<Integer, Integer> pair : map.entrySet()) {
    System.out.println(pair.getKey() + pair.getValue());
}
```

3. Use **values()**

```
for (Integer v : map.values())
      System.out.println("value: " + v);
```

4. Using **keySet** and **foreach**

```
for (Integer key : map.keySet()) {
    System.out.println(key + map.get(key));
}
```

5. Using **keySet** and **iterator**

```
Iterator<Integer> it = map.keySet().iterator();
while (itr.hasNext()) {
    Integer key = it.next();
    System.out.println(key + map.get(key));
}
```

6. Using **forEach** from Java 8

```
map.forEach((k, v) -> System.out.println(k + v));
```

7. Using **for** and **Map.Entry**

```
for (Iterator<Map.Entry<Integer, Integer>> entries = map.entrySet().iterator(); entries.hasNext(); ) {
    Map.Entry<Integer, Integer> entry = entries.next();
    System.out.println(entry.getKey() + entry.getValue());
}
```

8. Using the Java 8 **Stream API**

```
map.entrySet().stream().forEach(e -> System.out.println(e.getKey() + e.getValue());
```

6. What are differences between HashMap vs IdentityHashMap in java?

Answer

Property	java.util.HashMap	java.util.IdentityHashMap
Keys comparison (object-equality vs reference-equality)	Using object equality to compare the key and values. (equals() method)	Using reference equality to compare the key and values. (Operator “==”)
Initial size	16	21
Introduced in which java version	JDK 1.2	JDK 1.4
Program	Performs object-equality. o1 and o2 are equal iff (o1==null ? o2==null : o1.equals(o2)).	Performs reference-equality. o1 and o2 are equal iff (o1==o2).
Overridden equals() and hashCode() method call?	Overridden equals() - to compare. Use hashCode() - to find bucket location	Didn't overridde equals(), use "==". Didn't use hashCode(), instead uses System.identityHashCode()
Application - can maintain proxy object	Cannot maintain proxy object.	Can maintain proxy object in case need for object debugging.

7. What are differences between HashMap and ConcurrentHashMap in java?

Answer

Property	java.util.HashMap	java.util.concurrent. ConcurrentHashMap
Synchronization	Non-synchronized	Synchronized
2 threads on same Map object can access it at concurrently?	Yes, because it’s not synchronized (but it is not thread safe).	Yes, but depends on the **segments** based on concurrency level Multiple threads can access different segments concurrently.
Performance	**In a single-threaded environment**, HashMap performance is better as there is no synchronization. **In a multi-threaded environment**, HashMap can't allow more than one thread to access map.	**In a single-threaded environment,** ConcurrentHashMap performance is low as threads are required to wait. **In a multi-threaded environment,** ConcurrentHashMap has improved performance than Synchronized HashMap.
Null keys and values	Null key and null values are allowed.	Null key and null value are not allowed (NullPointerException)
Iterators	hashMap.keySet().iterator(); hashMap.values().iterator(); hashMap.entrySet().iterator(); Iterator is fail-fast, throws ConcurrentModificationException if object is modified after the iterator is created.	concurrentHashMap.keySet().iterator(); concurrentHashMap.values().iterator(); concurrentHashMap.entrySet().iterator(); Iterator is fail-safe and it will never throw ConcurrentModificationException.
putIfAbsent	Return **current value** associated with the provided key. Return **null** (if there was no mapping with the provided key before or it was mapped to a null value).	Return **current value** associated with the provided key. Return **null** (if there was no mapping with the provided key before or it was mapped to a null value).
Introduced in which java version	Java 2 (JDK 1.2)	Java 5 (JDK 1.5)
Implements which interface	java.util.Map	java.util.Map java.util.concurrent.ConcurrentMap
Package	java.util package	java.util.concurrent package

8. When to use HashMap vs Hashtable vs LinkedHashMap ?

Answer

Property	HashMap	Hashtable	LinkedHashMap
Insertion order	No guaranteed order, will remain constant over time	No guaranteed order, will remain constant over time	Sorted according to insertion-order
Performance	Faster than HashTable as it's non-synchronized	Slower than HashMap as Hashtable is synchronized	Time and space overhead is there because for maintaining order it internally uses Doubly Linked list.
Null keys and values	Allow one null key and more than one null values	Not allow to store null key or null value.	Allow one null key and more than one null values
Implements which interface	java.util.Map	java.util.Map	java.util.Map
Implementation uses?	Lists of Buckets	Buckets	Double-linked list of buckets
Complexity of put, get and remove methods	O(1)	O(1)	O(1)
Extends java.util.Dictionary (Abstract class, which is obsolete)	Doesn’t extends Dictionary	Hashtable extends Dictionary	Doesn’t extends Dictionary
Introduced in which java version?	Java 2 (JDK 1.2)	First version of Java (JDK 1.0)	Java 4 (JDK 1.4)