JAVA - THE WEAKHASHMAP CLASS

WeakHashMap is an implementation of the Map interface that stores only weak references to its keys. Storing only weak references allows a key-value pair to be garbagecollected when its key is no longer referenced outside of the WeakHashMap.

This class provides the easiest way to harness the power of weak references. It is useful for implementing "registry-like" data structures, where the utility of an entry vanishes when its key is no longer reachable by any thread.

The WeakHashMap functions identically to the HashMap with one very important exception: if the Java memory manager no longer has a strong reference to the object specified as a key, then the entry in the map will be removed.

Weak Reference: If the only references to an object are weak references, the garbage collector can reclaim the object's memory at any time.it doesn't have to wait until the system runs out of memory. Usually, it will be freed the next time the garbage collector runs.

The WeakHashMap class supports four constructors. The first form constructs a new, empty WeakHashMap with the default initial capacity (16) and the default load factor (0.75):

```
WeakHashMap()
```

The second form constructs a new, empty WeakHashMap with the given initial capacity and the default load factor, which is 0.75:

```
WeakHashMap(int initialCapacity)
```

The third form constructs a new, empty WeakHashMap with the given initial capacity and the given load factor.

```
WeakHashMap(int initialCapacity, float loadFactor)
```

The fourth form constructs a new WeakHashMap with the same mappings as the specified Map:

```
WeakHashMap(Map t)
```

Apart from the methods inherited from its parent classes, TreeMap defines the following methods:

SN	Methods with Description
1	**void clear()** Removes all mappings from this map.
2	**boolean containsKey(Object key)** Returns true if this map contains a mapping for the specified key.
3	**boolean containsValue(Object value)** Returns true if this map maps one or more keys to the specified value.
4	**Set entrySet()** Returns a collection view of the mappings contained in this map

5 **Object get(Object key)**

Returns the value to which the specified key is mapped in this weak hash map, or null if the map contains no mapping for this key.

6 **boolean isEmpty()**

Returns true if this map contains no key-value mappings.

7 **Set keySet()**

Returns a set view of the keys contained in this map.

8 **Object put(Object key, Object value)**

Associates the specified value with the specified key in this map.

9 **void putAll(Map m)**

Copies all of the mappings from the specified map to this map These mappings will replace any mappings that this map had for any of the keys currently in the specified map.

10 **Object remove(Object key)**

Removes the mapping for this key from this map if present.

11 **int size()**

Returns the number of key-value mappings in this map.

12 **Collection values()**

Returns a collection view of the values contained in this map.

Example:

The following program illustrates several of the methods supported by this collection:

```
import java.util.*;

public class WeakHashMap {
   private static Map map;
   public static void main (String args[]) {
      map = new WeakHashMap();
      map.put(new String("Maine"), "Augusta");

      Runnable runner = new Runnable() {
         public void run() {
            while (map.containsKey("Maine")) {
               try {
                  Thread.sleep(500);
               } catch (InterruptedException ignored) {
               }
               System.out.println("Thread waiting");
               System.gc();
            }
         }
      };
      Thread t = new Thread(runner);
      t.start();
      System.out.println("Main waiting");
```

```
      try {
         t.join();
      } catch (InterruptedException ignored) {
      }
   }
}
```

This would produce the following result:

```
Main waiting
Thread waiting
```

If you do not include the call to System.gc(), the system may never run the garbage collector as not much memory is used by the program. For a more active program, the call would be unnecessary.