java.util

Class HashMap

```
java.lang.Object
  |
  +--java.util.AbstractMap
        |
        +--java.util.HashMap
```

All Implemented Interfaces:
Cloneable, Map, Serializable

---

public class **HashMap**
extends AbstractMap
implements Map, Cloneable, Serializable

Hash table based implementation of the `Map` interface. This implementation provides all of the optional map operations, and permits `null` values and the `null` key. (The `HashMap` class is roughly equivalent to `Hashtable`, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

This implementation provides constant-time performance for the basic operations (`get` and `put`), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the `HashMap` instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the intial capacity too high (or the load factor too low) if iteration performance is important.

An instance of `HashMap` has two parameters that affect its performance: *initial capacity* and *load factor*. The *capacity* is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The *load factor* is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the `rehash` method.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (reflected in most of the operations of the `HashMap` class, including `get` and `put`). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of `rehash` operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, no `rehash` operations will ever occur.

If many mappings are to be stored in a `HashMap` instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.

Note that this implementation is not synchronized. If multiple threads access this map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be "wrapped" using the `Collections.synchronizedMap` method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
Map m = Collections.synchronizedMap(new HashMap(...));
```

The iterators returned by all of this class's "collection view methods" are *fail-fast*: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own `remove` or `add` methods, the iterator will throw a `ConcurrentModificationException`. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Since:

1.2

See Also:

`Object.hashCode()`, `Collection`, `Map`, `TreeMap`, `Hashtable`, Serialized Form

Inner classes inherited from class java.util.Map
`Map.Entry`

Constructor Summary

`HashMap``()` Constructs a new, empty map with a default capacity and load factor, which is `0.75`.
`HashMap``(int initialCapacity)` Constructs a new, empty map with the specified initial capacity and default load factor, which is `0.75`.
`HashMap``(int initialCapacity, float loadFactor)` Constructs a new, empty map with the specified initial capacity and the specified load factor.
`HashMap``(Map t)` Constructs a new map with the same mappings as the given map.

Method Summary

void	**clear**() Removes all mappings from this map.
Object	**clone**() Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
boolean	**containsKey**(Object key) Returns true if this map contains a mapping for the specified key.
boolean	**containsValue**(Object value) Returns true if this map maps one or more keys to the specified value.
Set	**entrySet**() Returns a collection view of the mappings contained in this map.
Object	**get**(Object key) Returns the value to which this map maps the specified key.
boolean	**isEmpty**() Returns true if this map contains no key-value mappings.
Set	**keySet**() Returns a set view of the keys contained in this map.
Object	**put**(Object key, Object value) Associates the specified value with the specified key in this map.
void	**putAll**(Map t) Copies all of the mappings from the specified map to this one.
Object	**remove**(Object key) Removes the mapping for this key from this map if present.
int	**size**() Returns the number of key-value mappings in this map.
Collection	**values**() Returns a collection view of the values contained in this map.

Methods inherited from class java.util.AbstractMap
equals, hashCode, toString

Methods inherited from class java.lang.Object
finalize, getClass, notify, notifyAll, wait, wait, wait

Methods inherited from interface java.util.Map
equals, hashCode