HashMap

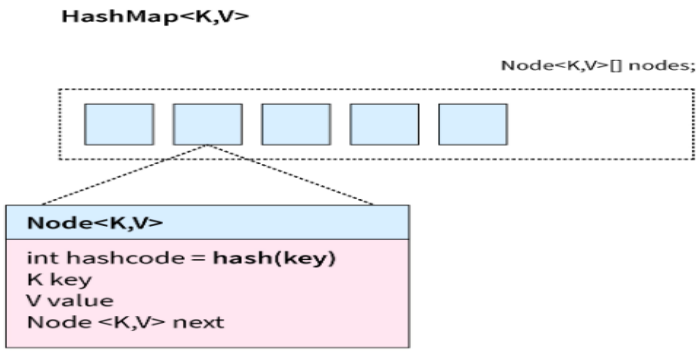
Overview

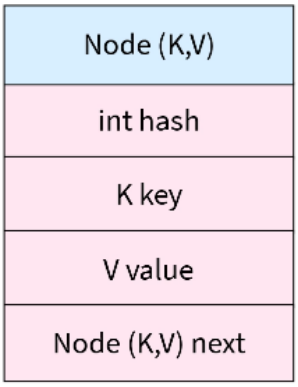
- The Underlying Data Structure is Hashtable.
- Duplicate Keys are not allowed.
- But Duplicate Values are allowed.
- Heterogeneous Objects are allowed for Both Keys and Values.
- Insertion Order is not preserved.
- It is based on the hash code of the keys.

1. Found in java.util package.
2. provides the basic implementation of the Map interface of Java.
3. Stores Key-value pairs.
4. HashMap is unsynchronised, therefore it's faster and uses less memory than HashTable. #Note: HashTable has been deprecated since Java 1.8.

Internal working of HashMap

- It is **based on** the principle of **hashing**.
- It is an algorithm to map an object to some representative integer values.
- Hashing is a process of **converting an object into integer** form by **using** the method **hashCode().**
- **hashCode()** method **returns memory reference of object in the integer form.**
- And that **integer form value maps with the key** to calculate the index.
- **HashMap has an internal Node<K,V> class.**
- HashMap<K,V> manages an array of Node<K,V>**.**
- Node<K,V> class consists of 4 fields
    - int hashcode = hash(key);
    - K key
    - V value
    - Node<K, V> next

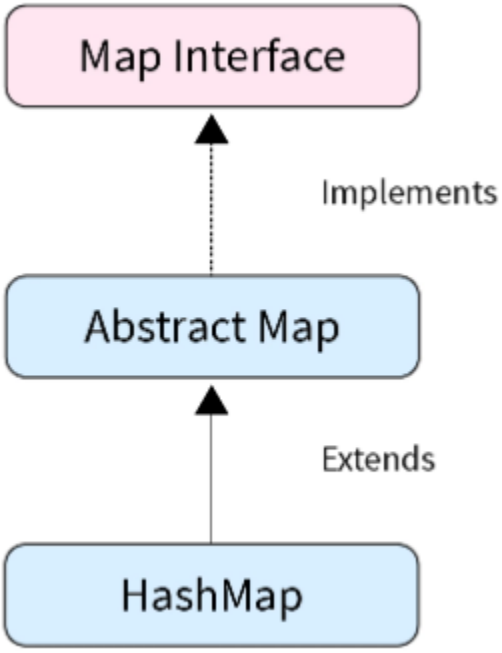




Rehashing

- Rehashing is the process of re-calculating the hash code of already stored entries.
- When the number of entries in the hash table exceeds the threshold value, the Map is rehashed so that it has approximately twice the number of buckets as before.
- e.g.,
  Let's suppose a HashMap is created with the initial default capacity of 16
  and the default load factor of 0.75.
  So, the threshold is 16 * 0.75 = 12,
  which means that it will increase the capacity from 16 to 32
  after the 12th entry (key-value pair) is added.
  This will be done by rehashing.

Hierarchy of HashMap



Declaration of HashMap

public class **HashMap**<K,V> extends **AbstractMap**<K,V> **implements Map<K,V>, Cloneable, Serializable**

Create HashMap Syntax

HashMap<K, V> hashmap = new HashMap<>();

e.g., HashMap<String, Integer> language= new HashMap<>();

Constructors in HashMap

1. HashMap() -
    - It is the default constructor.
    - HashMap instance with a default initial capacity of 16.
    - load factor is 0.75.
    - e.g., HashMap<String, Integer> language= new HashMap<>();
2. HashMap(int initialCapacity) -
    - It creates an instance of HashMap with a specified initial capacity.

- Load factor is 0.75.
- e.g., HashMap<Integer, String> hm1 = new HashMap<>(10);
- If threshold value(0.75) is exceeded then hm1 capacity will increase to 20. ie., will add 10 more memory sizes to existing HashMap.

3. HashMap(int initialCapacity, float loadFactor)
    - It creates an instance of HashMap with a specified initial capacity and specified load factor.
    - e.g., HashMap<Integer, String> hm1 = new HashMap<>(10, 0.5f);
    - New threshold=(5∗0.5)=2.5, if threshold value is exceeded then more 5 index will add to the capacity. It will become 10.
4. HashMap(Map map)
    - It creates an instance of HashMap using another Map.
    - e.g., HashMap<K, V> hm = new HashMap<K, V>(Map map);
    - hm and map both will have the same values, capacity and load factor.

1	V put(K key, V value)	Associates the specified value with the specified key in this map.
2	void putAll(Map<? extends K,? extends V> m)	Copies all of the mappings from the specified map to this map.
3	V putIfAbsent(K key, V value)	If the specified key is not already associated with a value (or is mapped to null), associates it with the given value and returns null, else returns the current value.
4	V get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
5	V getOrDefault(Object key, V defaultValue)	Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
6	Set keySet()	Returns a Set view of the keys contained in this map.
7	Set entrySet()	Returns a Set view of the mappings contained in this map.
8	V remove(Object key)	Removes the mapping for the specified key from this map if present.
9	boolean isEmpty()	Returns true if this map contains no key-value mappings.
10	boolean containsKey(Object key)	Returns true if this map contains a mapping for the specified key.
11	boolean containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
12	void clear()	Removes all of the mappings from this map.
13	Object clone()	Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.

14	boolean remove(Object key, Object value)	Removes the entry for the specified key only if it is currently mapped to the specified value.
15	V replace(K key, V value)	Replaces the entry for the specified key only if it is currently mapped to some value.
16	boolean replace(K key, V oldValue, V newValue)	Replaces the entry for the specified key only if currently mapped to the specified value.
17	void replaceAll(BiFunction<? super K, ? super V,? extends V> function)	Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
18	int size()	Returns the number of key-value mappings in this map.
19	Collection values()	Returns a Collection view of the values contained in this map.
20	void forEach(BiConsumer<? super K, ? super V> action)	Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
21	V compute(K key, BiFunction<? super K, ? super V,? extends V> remappingFunction)	Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
22	V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)	If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function and enters it into this map unless null.
23	V computeIfPresent(K key, BiFunction<? super K, ? super V,? extends V> remappingFunction)	If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
24	V merge(K key, V value, BiFunction<? super V, ? super V,? extends V> remappingFunction)	If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.

LinkedHashMap

- It is the Child Class of HashMap.
- It is Exactly Same as HashMap Except the following differences.

HashMap	LinkedHashMap
The Underlying Data Structure is Hashtable.	The Underlying Data Structure is Combination of Hashtable and LinkedList.
Insertion is Not Preserved	Insertion Order is Preserved.
Introduced in 1.2 Version	Introduced in 1.4 Version.

IdentityHashMap

- It is exactly the same as HashMap except the following Difference.
    - **In HashMap**, JVM will **use .equals() to Identify Duplicate Keys**, which is Meant for Content Comparison.
    - **In IdentityHashMap**, JVM will **use == Operator to Identify Duplicate Keys, which is Meant for Reference Comparison.**

```
IdentityHashMap<String, Integer> im = new IdentityHashMap<String, Integer>();
        String s1 = new String("10");
        String s2 = new String("10");
        im.put(s1, 10);
        im.put(s2, 20);
        for(Entry<String, Integer> i : im.entrySet()) {
                System.out.println(i);
        }
```

Output :

```
10=10
10=20
```

WeakHashMap

It is Exactly Same as HashMap Except the following Difference.

- In Case of **HashMap**, HashMap Dominates Garbage Collector. That is **if Object doesn't have any Reference** Still it is **Not Eligible for Garbage Collector** if it is associated with HashMap.
- But In Case of **WeakHashMap** if an **Object doesn't contain any References then it is Always Eligible for GC** Even though it is associated with WeakHashMap. That is, Garbage Collector Dominates WeakHashMap.

SortedMap

- It is the **Child Interface of Map**.
- If we want to Represent a Group of Key - Value Pairs According Some **Sorting Order** of keys then we should go for SortedMap.
- Specific methods defined in SortedMap
    1) Object **firstKey**();
    2) Object **lastKey**();
    3) SortedMap **headMap**(Object key)
    4) SortedMap **tailMap**(Object key)
    5) SortedMap **subMap**(Object key1, Object key2)
    6) Comparator **comparator**()

TreeMap

- The Underlying Data Structure is Red -Black Tree
- Duplicate Keys are Not Allowed. But Values can be Duplicated.
- Insertion Order is Not Preserved and it is Based on Some Sorting Order of Keys.
- If we are depending on Default Natural Sorting Order then the Keys should be Homogeneous and Comparable. Otherwise we will get a Runtime Exception Saying ClassCastException.
- If we define Our Own Sorting by Comparator then Keys can be Heterogeneous and NonComparable.
- But there are No Restrictions on Values. They can be Heterogeneous and Non-Comparable.

Null Acceptance

- For Empty TreeMap as the 1st Entry with null Key is Allowed.
- But After inserting that Entry if we are trying to Insert any Other Entry we will get RE: NullPointerException
- For Non- Empty TreeMap if we are trying to Insert null Entry then we will get Runtime Exception Saying NullPointerException
- There are No Restrictions on null Values.

Constructors:

```
1) TreeMap t = new TreeMap();                  // For Default Natural Sorting Order.
2) TreeMap t = new TreeMap(Comparator c);      // For Customized Sorting Order.
3) TreeMap t = new TreeMap(SortedMap m);       // InterConversion between Map Objects.
4) TreeMap t = new TreeMap(Map m);
```

HashTable

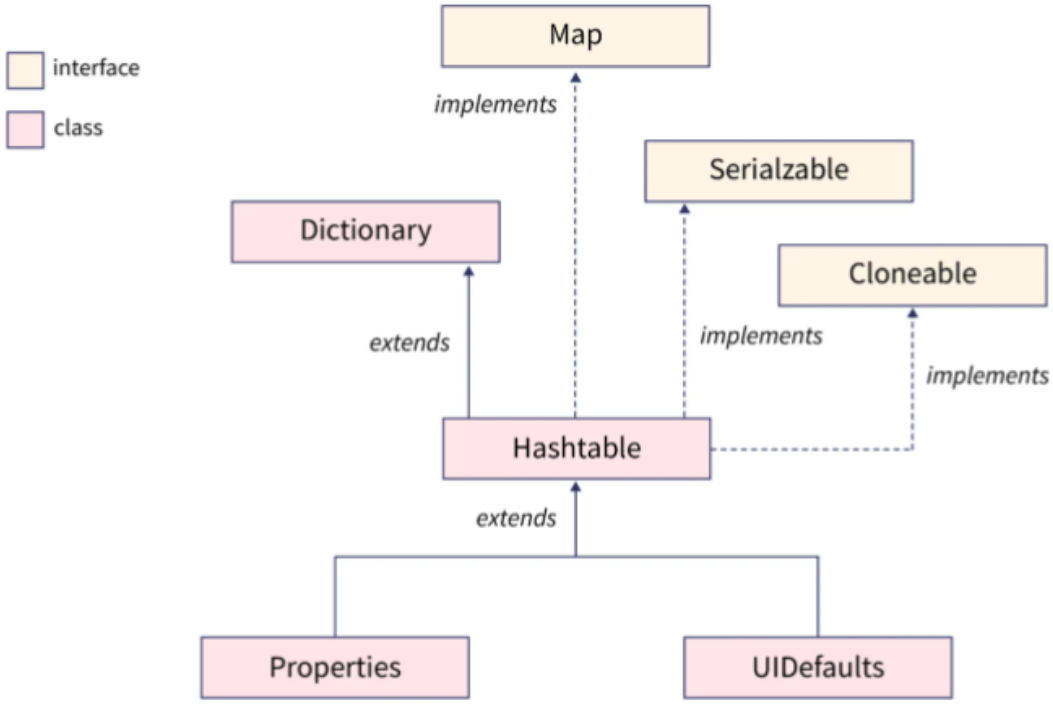
- Underlying **data structure is Hashtable** only.
- **Duplicate keys are not allowed.** But **values** can be **duplicated**.
- **Insertion order is not preserved** and it's **based on the hashcode** of the key.
- **Heterogenous keys and values are allowed**.
- **Null insertion is not allowed for keys and values** otherwise we will get an exception NullPointerException.
- Every **method** in the hashtable **is Synchronized**. Hence **Hashtable objects are thread safe.**
- Considered a "**legacy class**".

Declaration

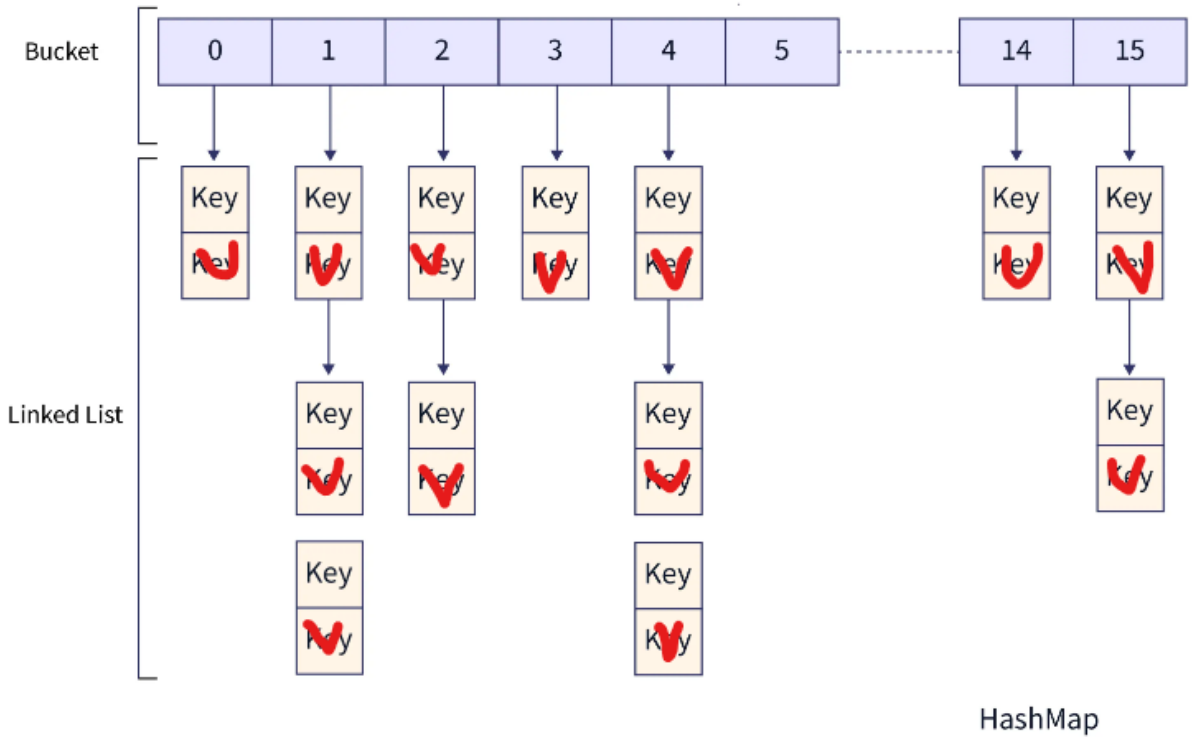
Java

```
public class Hashtable<K,V>

    extends Dictionary<K,V>

         implements Map<K,V>, Cloneable, Serializable
```

Internal Working



- A Hashtable is an array of a list.
- Each list referred to a bucket. And that list contains key/value pairs.
- It uses the hashCode() method for identifying which bucket should be assigned to the key/value combination.
- The equals() function is used by the hashtable to detect if two items are equal or not.



- Each entry in an array is called a bucket.
- Each value is linked with a bucket value by following formula
  Hash_number % Number of buckets = Index of Bucket 12 % 10 == [2]
- Multiple data values may collide at and be linked from the same bucket.
  eg., 12%10 = 2 , and 42%10 = 2
- Most hash tables manage collisions by comparing the whole value of a value being searched or added to each value existing in the linked list at the hashed-to bucket.This results in somewhat decreased speed but no functional confusion

Constructors

1. HashTable h = new HashTable();
   a. It is the default constructor of a hash table that constructs a new and empty hashtable with a default initial capacity (11) and load factor (0.75).

2. Hashtable h = new Hashtable(int initialcapacity);

a. It constructs a new, empty hashtable with the specified initial capacity and default load factor (0.75).

3. Hashtable h = new Hashtable(int initialcapacity, float fillRatio);
    a. It constructs a new, empty hashtable with the specified initial capacity and the specified load factor.

4. Hashtable h = new Hashtable(Map m);

    a. It constructs a new hashtable with the same mappings as the given Map.

Methods Present in HashTable

1	Object put(Object key, Object value)	**add the specified key to the specified value in this hashtable.**
2	putAll(Map<? extends K,? extends V> t)	**Copies all of the mappings from a given map to the hashtable.**
3	putIfAbsent(K key, V value)	**If the specified key is not already mapped with a value, maps it with the given value and returns null, else returns the current mapped value.**
4	Object get(Object key)	**Returns the value to which the specified key is mapped,** or null if the hashtable contains no mapping for the key.
5	getOrDefault(Object key, V defaultValue)	**Returns the value to which the given key is mapped, or defaultValue if the hashtable contains no mapping for the key.**
6	void clear()	**Removes all the key-value mappings** from a Hashtable and **makes it empty.**
7	Object clone()	Creates a **shallow copy of the hashtable. All the structure of the hashtable itself is copied, but the keys and values are not cloned.** This is a relatively expensive operation.
8	boolean containsValue(Object value)	**Tests if the specified object is a value in the hashtable.** Returns true if some value equal to the specified value exists within the hash table. Returns false if the value isn't found.
9	boolean containsKey(Object key)	**Tests if the specified object is a key in this hashtable.**
10	boolean contains(Object value)	**Tests if some key maps into the specified value in this hashtable.** This operation is more expensive than the containsKey method. Note that this method is identical in functionality to containsValue,

		(which is part of the Map interface in the collections framework)
11	boolean isEmpty()	**Tests if the hashtable maps no keys to values.**
12	void rehash()	**Increases the size of the hash table and rehashes all of its keys.**
13	Object remove(Object key)	**Removes the key** (and its corresponding value) from this hashtable.
14	int size()	**Returns the number of key-value mappings present in Hashtable**.
15	String toString()	Returns the objects of Hashtable in the form of a set of key-value mappings separated by ",". **The toString() method is used to convert all the elements of Hashtable into String.**
16	Enumeration keys()	**Returns an enumeration of the keys contained in the hash table.**
17	Enumeration elements()	**Returns an enumeration of the values contained in the hash table.**
18	hashcode()	**Returns the hash code value for the Map as per the definition in the Map interface.**
19	keySet()	**Returns a view of the keys contained in the hashtable as a set.**
20	entrySet()	**Returns a view of the mappings contained in the hashtable as a set.**