Some of the important points about HashMap in Java are;

* Java HashMap allows null key and null values.
* HashMap is not an ordered collection. You can iterate over HashMap entries through keys set but they are not guaranteed to be in the order of their addition to the HashMap.
* HashMap is almost similar to Hashtable except that it’s unsynchronized and allows null key and values.
* HashMap uses it’s inner class Node<K,V> for storing map entries.
* HashMap stores entries into multiple singly linked lists, called buckets or bins. Default number of bins is 16 and it’s always power of 2.
* HashMap uses hashCode() and equals() methods on keys for get and put operations. So HashMap key object should provide good implementation of these methods. This is the reason [immutable](https://www.journaldev.com/129/how-to-create-immutable-class-in-java) classes are better suitable for keys, for example String and Interger.
* Java HashMap is not thread safe, for multithreaded environment you should use ConcurrentHashMap class or get synchronized map using Collections.synchronizedMap() method.

**Java HashMap Constructors**

Java HashMap provides four constructors.

* **public HashMap()**: Most commonly used HashMap constructor. This constructor will create an empty HashMap with default initial capacity 16 and load factor 0.75
* **public HashMap(int initialCapacity)**: This HashMap constructor is used to specify the initial capacity and 0.75 load factor. This is useful in avoiding rehashing if you know the number of mappings to be stored in the HashMap.
* **public HashMap(int initialCapacity, float loadFactor)**: This HashMap constructor will create an empty HashMap with specified initial capacity and load factor. You can use this if you know the maximum number of mappings to be stored in HashMap. In common scenarios you should avoid this because load factor 0.75 offers a good tradeoff between space and time cost.
* **public HashMap(Map<? extends K, ? extends V> m)**: Creates a Map having same mappings as the specified map and with load factor 0.75

**Java HashMap Constructors Example**

Below code snippet is showing HashMap example of using all the above constructors.

Map<String, String> map1 = new HashMap<>();

Map<String, String> map2 = new HashMap<>(2^5);

Map<String, String> map3 = new HashMap<>(32,0.80f);

Map<String,String> map4 = new HashMap<>(map1);

**Java HashMap Methods**

Let’s have a look at the important methods of HashMap in java.

* **public void clear()**: This HashMap method will remove all the mappings and HashMap will become empty.
* **public boolean containsKey(Object key)**: This method returns ‘true’ if the key exists otherwise it will return ‘false’.
* **public boolean containsValue(Object value)**: This HashMap method returns true if the value exists otherwise false.
* **public Set<Map.Entry<K,V>> entrySet()**: This method returns a Set view of the HashMap mappings. This set is backed by the map, so changes to the map are reflected in the set, and vice-versa.
* **public V get(Object key)**: Returns the value mapped to the specified key, or null if there is no mapping for the key.
* **public boolean isEmpty()**: A utility method returning true if no key-value mappings are present.
* **public Set<K> keySet()**: Returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.
* **public V put(K key, V value)**: Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.
* **public void putAll(Map<? extends K, ? extends V> 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.
* **public V remove(Object key)**: Removes the mapping for the specified key from this map if present.
* **public int size()**: Returns the number of key-value mappings in this map.
* **public Collection<V> values()**: Returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa.

There are many new methods in HashMap introduced in [Java 8](https://www.journaldev.com/2389/java-8-features-with-examples).

* **public 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), this method attempts to compute its value using the given mapping function and enters it into the HashMap unless Null.
* **public 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.
* **public V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**: This HashMap method attempts to compute a mapping for the specified key and its current mapped value.
* **public void forEach(BiConsumer<? super K, ? super V> action)**: This method performs the given action for each entry in this map.
* **public V getOrDefault(Object key, V defaultValue)**: Same as get except that defaultValue is returned if no mapping found for the specified key.
* **public 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. Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is null.
* **public 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.
* **public boolean remove(Object key, Object value)**: Removes the entry for the specified key only if it is currently mapped to the specified value.
* **public boolean replace(K key, V oldValue, V newValue)**: Replaces the entry for the specified key only if currently mapped to the specified value.
* **public V replace(K key, V value)**: Replaces the entry for the specified key only if it is currently mapped to some value.
* **public 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.

package com.journaldev.examples;

import java.util.Collection;

import java.util.HashMap;

import java.util.Map;

import java.util.Map.Entry;

import java.util.Set;

public class HashMapExample {

public static void main(String[] args) {

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

map.put("1", "1"); // put example

map.put("2", "2");

map.put("3", "3");

map.put("4", null); // null value

map.put(null, "100"); // null key

String value = map.get("3"); // get example

System.out.println("Key = 3, Value = " + value);

value = map.getOrDefault("5", "Default Value");

System.out.println("Key = 5, Value=" + value);

boolean keyExists = map.containsKey(null);

boolean valueExists = map.containsValue("100");

System.out.println("keyExists=" + keyExists + ", valueExists=" + valueExists);

Set<Entry<String, String>> entrySet = map.entrySet();

System.out.println(entrySet);

System.out.println("map size=" + map.size());

Map<String, String> map1 = new HashMap<>();

map1.putAll(map);

System.out.println("map1 mappings= " + map1);

String nullKeyValue = map1.remove(null);

System.out.println("map1 null key value = " + nullKeyValue);

System.out.println("map1 after removing null key = " + map1);

Set<String> keySet = map.keySet();

System.out.println("map keys = " + keySet);

Collection<String> values = map.values();

System.out.println("map values = " + values);

map.clear();

System.out.println("map is empty=" + map.isEmpty());

}

}

Key = 3, Value = 3

Key = 5, Value=Default Value

keyExists=true, valueExists=true

[null=100, 1=1, 2=2, 3=3, 4=null]

map size=5

map1 mappings= {null=100, 1=1, 2=2, 3=3, 4=null}

map1 null key value = 100

map1 after removing null key = {1=1, 2=2, 3=3, 4=null}

map keys = [null, 1, 2, 3, 4]

map values = [100, 1, 2, 3, null]

map is empty=true

**Java HashMap Load Factor**

Load Factor is used to figure out when HashMap will be rehashed and bucket size will be increased. Default value of bucket or capacity is 16 and load factor is 0.75. Threshold for rehashing is calculated by multiplying capacity and load factor. So default threshold value will be 12. So when the HashMap will have more than 12 mappings, it will be rehashed and number of bins will be increased to next of power 2 i.e 32. Note that HashMap capacity is always power of 2.

Default load factor of 0.75 provides good tradeoff between space and time complexity. But you can set it to different values based on your requirement. If you want to save space, then you can increase it’s value to 0.80 or 0.90 but then get/put operations will take more time

**Java HashMap keySet**

package com.journaldev.examples;

import java.util.HashMap;

import java.util.HashSet;

import java.util.Map;

import java.util.Set;

public class HashMapKeySetExample {

public static void main(String[] args) {

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

map.put("1", "1");

map.put("2", "2");

map.put("3", "3");

Set<String> keySet = map.keySet();

System.out.println(keySet);

map.put("4", "4");

System.out.println(keySet); // keySet is backed by Map

keySet.remove("1");

System.out.println(map); // map is also modified

keySet = new HashSet<>(map.keySet()); // copies the key to new Set

map.put("5", "5");

System.out.println(keySet); // keySet is not modified

}

}

[1, 2, 3]

[1, 2, 3, 4]

{2=2, 3=3, 4=4}

[2, 3, 4]

**Java HashMap values**

package com.journaldev.examples;

import java.util.Collection;

import java.util.HashMap;

import java.util.Map;

public class HashMapValuesExample {

public static void main(String[] args) {

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

map.put("1", "1");

map.put("2", "2");

map.put("3", null);

map.put("4", null);

map.put(null, "100");

Collection<String> values = map.values();

System.out.println("map values = " + values);

map.remove(null);

System.out.println("map values after removing null key = " + values);

map.put("5", "5");

System.out.println("map values after put = " + values);

System.out.println(map);

values.remove("1"); // changing values collection

System.out.println(map); // updates in map too

}

}

map values = [100, 1, 2, null, null]

map values after removing null key = [1, 2, null, null]

map values after put = [1, 2, null, null, 5]

{1=1, 2=2, 3=null, 4=null, 5=5}

{2=2, 3=null, 4=null, 5=5}

**Java HashMap entrySet**

package com.journaldev.examples;

import java.util.AbstractMap;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import java.util.Map.Entry;

import java.util.Set;

public class HashMapEntrySetExample {

public static void main(String[] args) {

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

map.put("1", "1");

map.put("2", null);

map.put(null, "100");

Set<Entry<String,String>> entrySet = map.entrySet();

Iterator<Entry<String, String>> iterator = entrySet.iterator();

Entry<String, String> next = null;

System.out.println("map before processing = "+map);

System.out.println("entrySet before processing = "+entrySet);

while(iterator.hasNext()){

next = iterator.next();

System.out.println("Processing on: "+next.getValue());

if(next.getKey() == null) iterator.remove();

}

System.out.println("map after processing = "+map);

System.out.println("entrySet after processing = "+entrySet);

Entry<String, String> simpleEntry = new AbstractMap.SimpleEntry<String, String>("1","1");

entrySet.remove(simpleEntry);

System.out.println("map after removing Entry = "+map);

System.out.println("entrySet after removing Entry = "+entrySet);

}

}