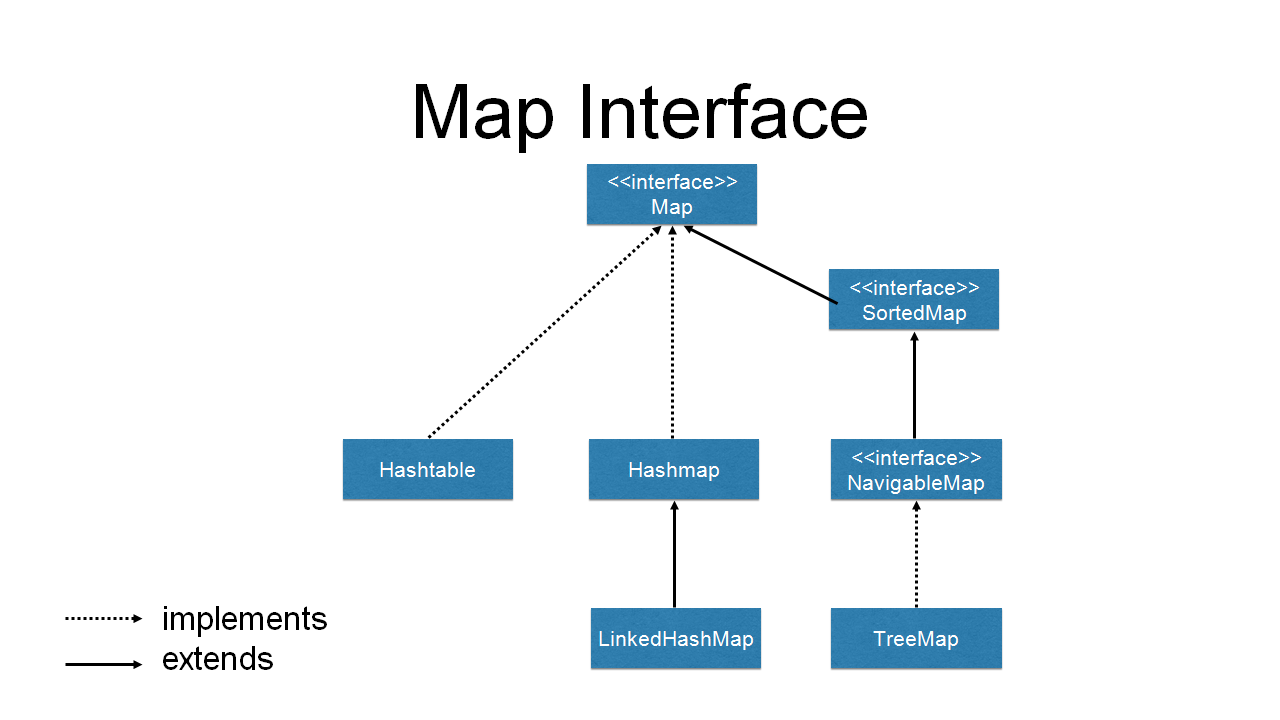
**Java Map Interface** java.util.Map



The Map interface of the Java collections framework provides the functionality of the map data structure.

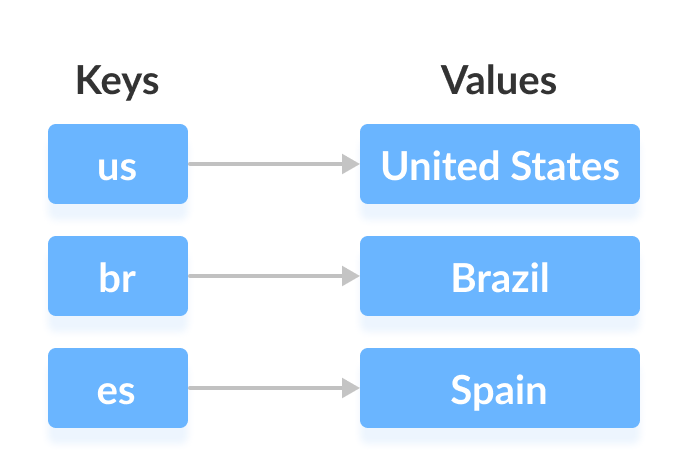
It implements the Collection interface.

A value does **not** have a specific position in a map but can be retrieved using the key it is paired with

## Working of Map

In Java, elements of Map are stored in **key/value** pairs. **Keys** are unique values associated with individual **Values**.

A map cannot contain duplicate keys. And, each key is associated with a single value.



We can access and modify values using the keys associated with them.

In the above diagram, we have values: United States, Brazil, and Spain. And we have corresponding keys: us, br, and es.

Now, we can access those values using their corresponding keys.

**Note:** The Map interface maintains 3 different sets:

* the set of keys
* the set of values
* the set of key/value associations (mapping).

Hence we can access keys, values, and associations individually.

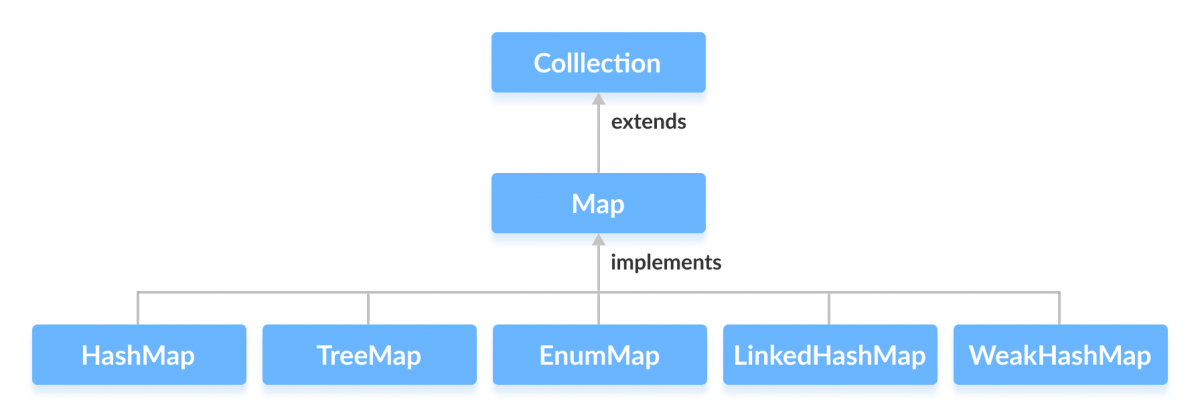
## **Classes that implement Map**

Since Map is an interface, we cannot create objects from it.

In order to use functionalities of the Map interface, we can use these classes:

* [HashMap](https://www.programiz.com/java-programming/hashmap)
* [EnumMap](https://www.programiz.com/java-programming/enummap)
* [LinkedHashMap](https://www.programiz.com/java-programming/linkedhashmap)
* [WeakHashMap](https://www.programiz.com/java-programming/weakhashmap)
* [TreeMap](https://www.programiz.com/java-programming/treemap)

These classes are defined in the collections framework and implement the Map interface.



## Interfaces that extend Map

The Map interface is also extended by these subinterfaces:

* [SortedMap](https://www.programiz.com/java-programming/sortedmap)
* [NavigableMap](https://www.programiz.com/java-programming/navigablemap)
* [ConcurrentMap](https://www.programiz.com/java-programming/concurrentmap)



## How to use Map?

In Java, we must import the java.util.Map package in order to use Map. Once we import the package, here's how we can create a map.

// Map implementation using HashMap

Map<Key, Value> numbers = new HashMap<>();

In the above code, we have created a Map named numbers. We have used the HashMap class to implement the Map interface.

Here,

* Key - a unique identifier used to associate each element (value) in a map
* Value - elements associated by keys in a map

## Methods of Map

The Map interface includes all the methods of the Collection interface. It is because Collection is a super interface of Map.

Besides methods available in the Collection interface, the Map interface also includes the following methods:

* **put(K, V)** - Inserts the association of a key K and a value V into the map. If the key is already present, the new value replaces the old value.
* **putAll()** - Inserts all the entries from the specified map to this map.
* **putIfAbsent(K, V)** - Inserts the association if the key K is not already associated with the value V.
* **get(K)** - Returns the value associated with the specified key K. If the key is not found, it returns null.
* **getOrDefault(K, defaultValue)** - Returns the value associated with the specified key K. If the key is not found, it returns the defaultValue.
* **containsKey(K)** - Checks if the specified key K is present in the map or not.
* **containsValue(V)** - Checks if the specified value V is present in the map or not.
* **replace(K, V)** - Replace the value of the key K with the new specified value V.
* **replace(K, oldValue, newValue)** - Replaces the value of the key K with the new value newValue only if the key K is associated with the value oldValue.
* **remove(K)** - Removes the entry from the map represented by the key K.
* **remove(K, V)** - Removes the entry from the map that has key K associated with value V.
* **keySet()** - Returns a set of all the keys present in a map.
* **values()** - Returns a set of all the values present in a map.
* **entrySet()** - Returns a set of all the key/value mapping present in a map.

## Properties Of Map Interface In Java :

1) Map interface is a part of Java Collection Framework, but it doesn’t inherit **Collection Interface**.

2) Map interface stores the data as a **key-value pairs** where each key is associated with a value.

3) A map can not have duplicate **keys** but can have duplicate **values**.

4) Each key **at most** must be associated with one value.

5) Each key-value pairs of the map are stored as **Map.Entry** objects. Map.Entry is an inner interface of Map interface.

6) The common implementations of Map interface are **HashMap**, **LinkedHashMap** and **TreeMap**.

7) Order of elements in map is implementation dependent. **HashMap** doesn’t maintain any order of elements. **LinkedHashMap** maintains **insertion order** of elements. Where as **TreeMap** places the elements according to supplied **Comparator**.

8) The Map interface provides three methods, which allows map’s contents to be viewed as a **set of keys**(keySet() method), **collection of values**(values() method), or **set of key-value mappings**(entrySet() method).

# The Map.Entry Interface

The Map.Entry interface enables you to work with a map entry.

The **entrySet( )** method declared by the Map interface returns a Set containing the map entries. Each of these set elements is a Map.Entry object.

## Method of Map.Entry interface

1) boolean equals(Object o): Compares the specified object with this entry for equality.  
2) Key getKey(): Returns the key corresponding to this entry.  
3) Value getValue(): Returns the value corresponding to this entry.  
4) int hashCode(): Returns the hash code value for this map entry.  
5) Value setValue(V value): Replaces the value corresponding to this entry with the specified value (optional operation).

# Hashtable in java with example

public class **Hashtable<K,V>**

extends [Dictionary](https://docs.oracle.com/javase/7/docs/api/java/util/Dictionary.html)<K,V>

implements [Map](https://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<K,V>, [Cloneable](https://docs.oracle.com/javase/7/docs/api/java/lang/Cloneable.html), [Serializable](https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html)

This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value.

Hashtable is similar to [HashMap](https://beginnersbook.com/2013/12/hashmap-in-java-with-example/) except it is synchronized.

A Hashtable internally contains buckets in which it stores the key/value pairs. The Hashtable uses the key's **hashcode** to determine to which bucket the key/value pair should map.

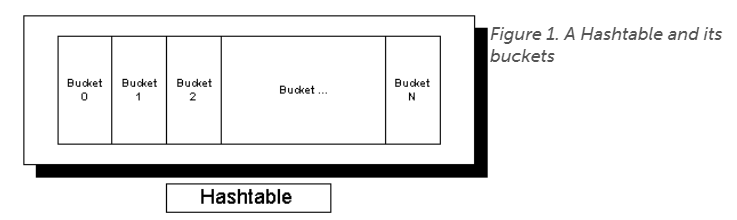


Figure 1 shows a Hashtable and its buckets. When you pass a key/value to the Hashtable, it queries the key's hashcode. The Hashtable uses that code to determine the bucket in which to place the key/value. So, for example, if the hashcode equals zero, the Hashtable places the value into Bucket 0. Likewise, if the hashcode is two, the Hashtable places the value into Bucket 2. (This is a simplistic example; the Hashtable will massage the hashcode first so the Hashtable doesn't try to insert the value outside the bucket.)

By using the hashcode this way, the Hashtable can also quickly determine in which bucket it has placed the value when you try to retrieve it.

Hashcodes, however, represent only half the picture. The hashcode only tells the Hashtable into which bucket to drop the key/value. Sometimes, however, multiple objects may map to the same bucket, an event known as a *collision.* In Java, the Hashtable responds to a collision by placing multiple values into the same bucket (other implementations may handle collisions differently). Figure 2 shows what a Hashtable might look like after a few collisions.

**Q:**When I use an object as a key in a Hashtable , what in the Object class must I override and why?

**A:**When you create your own key object for use in a Hashtable, you must override the [Object.equals()](http://www.javaworld.com/#resources) and [Object.hashCode()](http://www.javaworld.com/#resources) methods since Hashtable uses a combination of the key's hashCode() and equals() methods to store and retrieve its entries quickly. It's also a general rule that when you override equals(), you always override hashCode().

## Class HashMap<K,V>

* [java.lang.Object](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html)
  + [java.util.AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<K,V>
    - java.util.HashMap<K,V>
* **Type Parameters:**

K - the type of keys maintained by this map

V - the type of mapped values

**All Implemented Interfaces:**

[Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html), [Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html), [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<K,V>

**Direct Known Subclasses:**

[LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html), [PrinterStateReasons](https://docs.oracle.com/javase/8/docs/api/javax/print/attribute/standard/PrinterStateReasons.html)

public class **HashMap<K,V>**

extends [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html)<K,V>

implements [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html)<K,V>, [Cloneable](https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html), [Serializable](https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html)

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 initial 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 hash table is *rehashed* (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

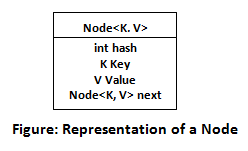
Note : HashMap when we have Key-Value pair and we use HashSet when we do not need repetitive data.

# Working of HashMap in Java

## **What is Hashing**

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.



* **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
* **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
* **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.

## **Insert Key, Value pair in HashMap**

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

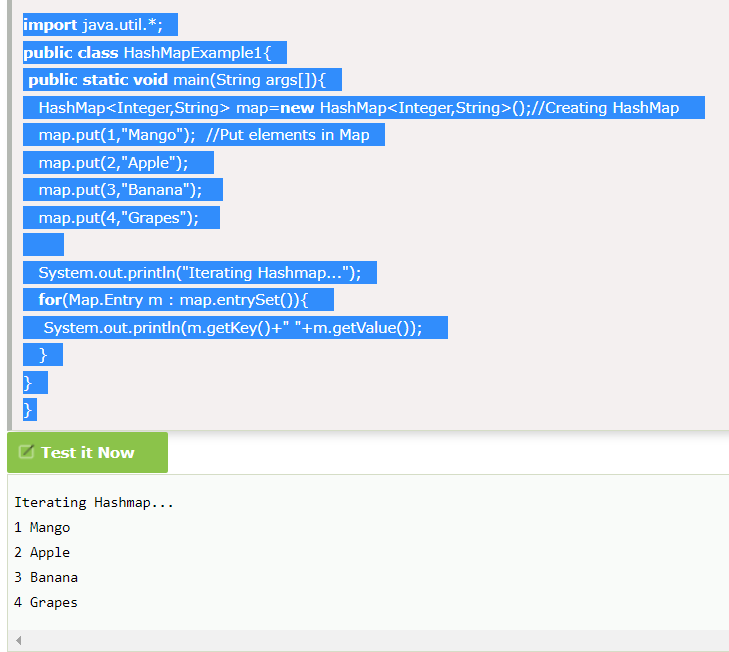
## **Hash Collision**

This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

1. Index=63281940 & (16-1) = 4

### **Points to remember**

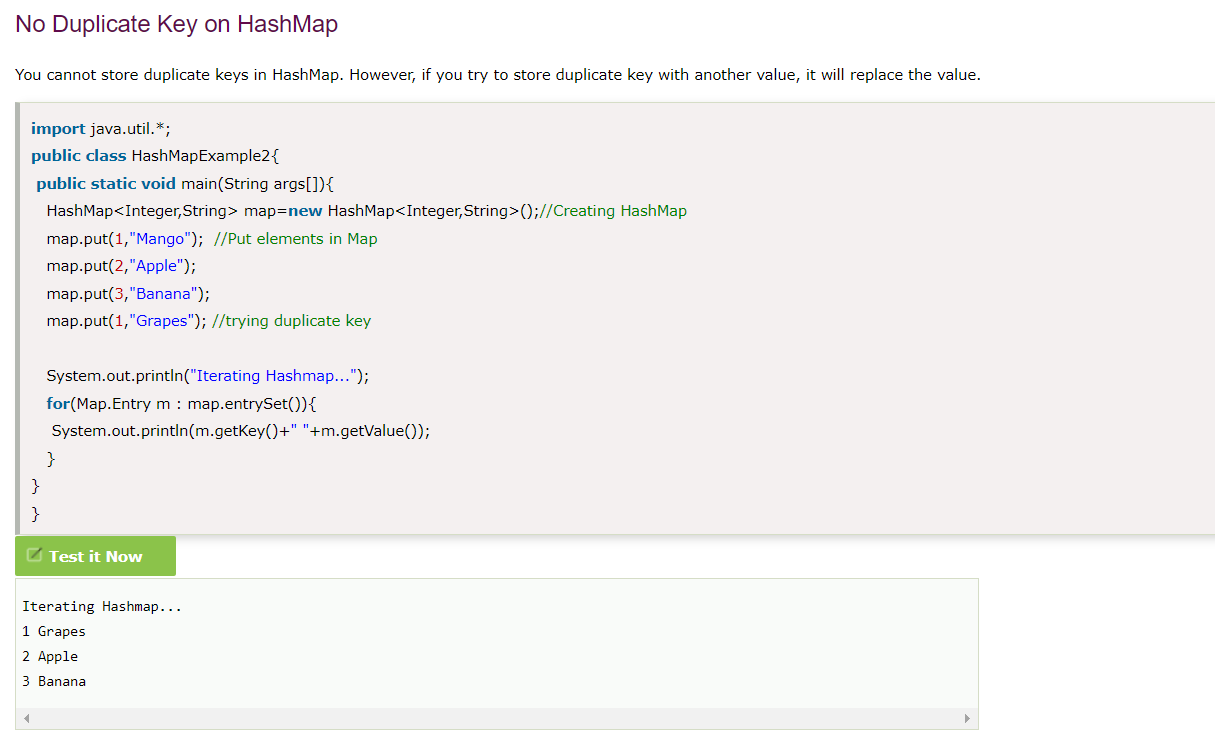
* Java HashMap contains values based on the key.
* Java HashMap contains only unique keys.
* Java HashMap may have one null key and multiple null values.
* Java HashMap is non synchronized.
* Java HashMap maintains no order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.



To get the key and value elements, we should call the getKey() and getValue() methods. The Map.Entry interface contains the getKey() and getValue() methods. But, we should call the entrySet() method of Map interface to get the instance of Map.Entry.

### **No Duplicate Key on HashMap**

You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the old value with the new value.



**How to Sort HashMap in Java**

Java HashMap does not preserve any order by default. If there is a need to sort HashMap we sort it explicitly based on the requirements. Java provides an option to sort HashMap based on keys and values.

* Sort HashMap by Keys
* Sort HashMap by Values

## **Sort HashMap by Keys**

There are following ways to sort HashMap by keys:

* By using **TreeMap**
* By using **LinkedHashMap**

When we use LinkedHashMap, we should follow the process:

When we use LinkedHashMap, then we need to get Key set. Convert the Set into List, sort the list and then add the sorted list into LinkedHashMap in the same order. The same process we have done in the example **Sort HashMap by Value**.