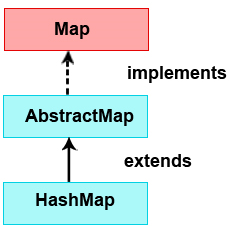
**1. HashMap Class Overview**

Java *HashMap* class implements the *Map* interface by using a hash table. It inherits the *AbstractMap* class and implements the *Map* interface.

****

The important points about the Java *HashMap* class:

* A HashMap cannot contain duplicate keys.
* Java HashMap allows null values and the single null key.
* HashMap is an unordered collection. It does not guarantee any specific order of the elements.
* Java HashMap is not thread-safe. You must explicitly synchronize concurrent modifications to the HashMap.

**2. Creating a HashMap and Adding Key-Value Pairs to It**

The following example shows how to create a *HashMap*, and add new key-value pairs to it.

// Creating a HashMap

Map<String, Integer> numberMapping = new HashMap<>();

// Adding key-value pairs to a HashMap

numberMapping.put("One", 1);

numberMapping.put("Two", 2);

numberMapping.put("Three", 3);

// Add a new key-value pair only if the key does not exist in the HashMap, or is mapped to `null`

numberMapping.putIfAbsent("Four", 4);

System.out.println(numberMapping);

**put(String key, Integer value)**

The put() method associates the specified value with the specified key in this map (optional operation).

// Creating a HashMap

Map<String, Integer> numberMapping = new HashMap<>();

// Adding key-value pairs to a HashMap

numberMapping.put("One", 1);

numberMapping.put("Two", 2);

numberMapping.put("Three", 3);

**putIfAbsent(String key, Integer value)**

The *putIfAbsent()* method of the *HashMap* class is used to map the specified key with the specified value, only if no such key exists (or is mapped to null) in this HashMap instance.

// Creating a HashMap

Map<String, Integer> numberMapping = new HashMap<>();

// Add a new key-value pair only if the key does not exist in the HashMap, or is mapped to `null`

numberMapping.putIfAbsent("Four", 4);

**3. Hashmap APIs for Accessing Keys and Modifying Their Associated Value**

* How to check if a *HashMap*is empty | *isEmpty()*
* How to find the size of a *HashMap*| *size()*
* How to check if a given key exists in a *HashMap*| *containsKey()*
* How to check if a given value exists in a *HashMap*| *containsValue()*
* How to get the value associated with a given key in the *HashMap*| *get()*
* How to modify the value associated with a given key in the *HashMap*| *put()*

**isEmpty()**

The isEmpty() method returns *true* if this map contains no key-value mappings.

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

// Check if a HashMap is empty

System.out.println("is userCityMapping empty? : " + userCityMapping.isEmpty());

**size()**

The size() method returns the number of key-value mappings in this map. If the map contains more than *Integer.MAX\_VALUE*elements return *Integer.MAX\_VALUE.*

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

userCityMapping.put("John", "New York");

userCityMapping.put("Rajeev", "Bengaluru");

userCityMapping.put("Steve", "London");

// Find the size of a HashMap

System.out.println("We have the city information of " + userCityMapping.size() + " users");

**containsKey(Object key)**

The containsKey() method returns *true* if this map contains a mapping for the specified key.

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

userCityMapping.put("John", "New York");

userCityMapping.put("Rajeev", "Bengaluru");

userCityMapping.put("Steve", "London");

String userName = "Steve";

// Check if a key exists in the HashMap

if(userCityMapping.containsKey(userName)) {

// Get the value assigned to a given key in the HashMap

String city = userCityMapping.get(userName);

System.out.println(userName + " lives in " + city);

} else {

System.out.println("City details not found for user " + userName);

}

**containsValue(Object value)**

The *containsValue()* method returns *true* if this map maps one or more keys to the specified value.

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

userCityMapping.put("John", "New York");

userCityMapping.put("Rajeev", "Bengaluru");

userCityMapping.put("Steve", "London");

// Check if a value exists in a HashMap

if(userCityMapping.containsValue("New York")) {

System.out.println("There is a user in the userCityMapping who lives in New York");

} else {

System.out.println("There is not user in the userCityMapping who lives in New York");

}

**get(Object key)**

The *get()* method returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

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

userCityMapping.put("John", "New York");

userCityMapping.put("Rajeev", "Bengaluru");

userCityMapping.put("Steve", "London");

System.out.println("Lisa's city : " + userCityMapping.get("Steve"));

**4. HashMap Remove APIs with Examples**

* Remove a key from a *HashMap*| *remove(Object key)*
* Remove a key from a *HashMap*only if it is associated with a given value | *remove(Object key, Object value)*

**remove(Object key)**

The remove() method removes the mapping for a key from this map if it is present.

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

husbandWifeMapping.put("Jack", "Marie");

husbandWifeMapping.put("Chris", "Lisa");

husbandWifeMapping.put("Steve", "Jennifer");

String husband = "Chris";

String wife = husbandWifeMapping.remove(husband);

**remove(Object key, Object value)**

The remove(Object key, Object value) method removes the entry for the specified key only if it is currently mapped to the specified value.

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

husbandWifeMapping.put("Jack", "Marie");

husbandWifeMapping.put("Chris", "Lisa");

husbandWifeMapping.put("Steve", "Jennifer");

// Remove a key from the HashMap only if it is mapped to the given value

// Ex - Divorce "Jack" only if He is married to "Linda"

boolean isRemoved = husbandWifeMapping.remove("Jack", "Linda");

System.out.println("Did Jack get removed from the mapping? : " + isRemoved);

**5. Hashmap Demonstration for Null Keys and Null Values**

// HashMap demonstration for null keys and null values

private static void nullKeyValueDemo() {

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

map.put(null, null);

map.put(null, null);

// iterate map using java 8 forEach method

map.forEach((k, v) -> {

System.out.println(k);

System.out.println(v);

});

for (Entry<String, String> entry : map.entrySet()) {

System.out.println(entry.getKey());

System.out.println(entry.getValue());

}

}

// HashMap demonstration for duplicate keys

private static void duplicateKeyDemo() {

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

map.put("key1", "value1");

map.put("key1", "value2");

// iterate map using java 8 forEach method

map.forEach((k, v) -> {

System.out.println(k);

System.out.println(v);

});

for (Entry<String, String> entry : map.entrySet()) {

System.out.println(entry.getKey());

System.out.println(entry.getValue());

}

}

**6. How to Perform Range View Operations in Map?**

The Collection view methods allow a Map to be viewed as a Collection in these three ways:

* *keySet*— the *Set*of keys contained in the Map.
* *values*— The *Collection*of values contained in the *Map*. This *Collection*is not a *Set*, because multiple keys can map to the same value.
* *entrySet*— the Set of key-value pairs contained in the Map. The *Map*interface provides a small nested interface called *Map.Entry,* the type of the elements in this Set.

private static void collectionViewsDemo() {

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

map.put("key1", "value1");

map.put("key2", "value2");

map.put("key3", "value3");

// Returns a Set view of the keys contained in this map

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

// Returns a Collection view of the values contained in this map

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

// Returns a Set view of the mappings contained in this map

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

// iterate map using java 8 forEach method

map.forEach((k, v) -> {

System.out.println(k);

System.out.println(v);

});

for (Entry<String, String> pair : entry) {

System.out.println(pair.getKey());

System.out.println(pair.getValue());

}

}

**7. Different Ways to Iterate Over Map**

private static void iterateMap() {

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

map.put("key1", "value1");

map.put("key2", "value2");

map.put("key3", "value3");

// Returns a Set view of the keys contained in this map

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

// Returns a Collection view of the values contained in this map

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

// Returns a Set view of the mappings contained in this map

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

for (Entry<String, String> pair : entry) {

System.out.println(pair.getKey());

System.out.println(pair.getValue());

}

// iterate map using java 8 forEach method

map.forEach((k, v) -> {

System.out.println(k);

System.out.println(v);

});

}

**8. How to Store Multiple Values in Map?**

A multimap is like a Map but it can map each key to multiple values.

Let's create a list of objects and map with the key.

private static void multmapDemo() {

Map<String, List<String>> multimap = new HashMap<>();

List<String> multiValueList = new ArrayList<>();

multiValueList.add("value1");

multiValueList.add("value2");

multiValueList.add("value3");

multimap.put("key1", multiValueList);

}

**9. Java 8 ForEach() Method With Map**

The normal way to loop a *Map*.

public static void forEachWithMap() {

// Before Java 8, how to loop map

final Map<Integer, Person> map = new HashMap<>();

map.put(1, new Person(100, "Ramesh"));

map.put(2, new Person(100, "Ram"));

map.put(3, new Person(100, "Prakash"));

map.put(4, new Person(100, "Amir"));

map.put(5, new Person(100, "Sharuk"));

for (final Entry<Integer, Person> entry : map.entrySet()) {

System.out.println(entry.getKey());

System.out.println(entry.getValue().getName());

}

}

In Java 8, you can loop a *Map*with *forEach*and lambda expressions.

public static void forEachWithMap() {

// Before Java 8, how to loop map

final Map<Integer, Person> map = new HashMap<>();

map.put(1, new Person(100, "Ramesh"));

map.put(2, new Person(100, "Ram"));

map.put(3, new Person(100, "Prakash"));

map.put(4, new Person(100, "Amir"));

map.put(5, new Person(100, "Sharuk"));

// In Java 8, you can loop a Map with forEach + lambda expression.

map.forEach((k,p) -> {

System.out.println(k);

System.out.println(p.getName());

});

}

**10. Synchronizing Access to Java HashMap**

HashMap is not synchronized, which means you cannot use it on a multi-threaded Java program without external synchronization.

In another word, if you share one instance of *HashMap* between multiple threads, each is either adding, removing, or updating entries then it's possible that *HashMap* loses its structure and does not behave as expected.

* Use the *Collections.synchronizedMap()* method to obtain a synchronized view of the HashMap.
* Write the increment logic inside a *synchronized* block.
* We can use a ConcurrentHashMap for thread safety instead of the HashMap obtained via *Collections.synchronizedMap()* method. The ConcurrentHashMap provides thread-safe operations on the Map.

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

currencies.put("USA", "USD");

currencies.put("England", "GBP");

currencies.put("Canada", "CAD");

currencies.put("HongKong", "HKD");

currencies.put("Australia", "AUD");

// Synchronizing HashMap in Java

currencies = Collections.synchronizedMap(currencies);

// Make sure to synchronize Map while Iterating

// getting key set can be outside synchronized block

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

synchronized (currencies) {

Iterator<String> itr = keySet.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

}

}