**Java HashMap**

The HashMap class of the Java collections framework provides the hash table implementation of the Map interface.

## Create a HashMap

In order to create a hash map, we must import the java.util.HashMap package first. Once we import the package, here is how we can create hashmaps in Java.

// HashMap creation with 8 capacity and 0.6 load factor

HashMap<Key, Value> numbers = new HashMap<>(8, 0.6f);

In the above code, we have created a hashmap named numbers.

Here,

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

Notice the part new HashMap<>(8, 0.6). Here, the first parameter is **capacity** and the second parameter is **loadFactor**.

* **capacity** - The capacity of this hash map is 8. Meaning, it can store 8 entries.
* **loadFactor** - The load factor of this hashmap is 0.6. This means, whenever our hash table is filled by 60%, the entries are moved to a new hash table of double the size of the original hash table.

**Default capacity and load factor**

It's possible to create a hashmap without defining its capacity and load factor. For example,

// HashMap with default capacity and load factor

HashMap<Key, Value> numbers1 = new HashMap<>();

By default,

* the capacity of the hash map will be 16
* the load factor will be 0.75

## Creating HashMap from Other Maps

Here is how we can create a hashmap containing all the elements of other maps.

import java.util.HashMap;

class Main {

public static void main(String[] args) {

// Creating a hashmap of even numbers

HashMap<String, Integer> evenNumbers = new HashMap<>();

evenNumbers.put("Two", 2);

evenNumbers.put("Four", 4);

System.out.println("HashMap1: " + evenNumbers);

// Creating a hash map from other hashmap

HashMap<String, Integer> numbers = new HashMap<>(evenNumbers);

numbers.put("Three", 3);

System.out.println("HashMap2: " + numbers);

}

}

**Output**

HashMap1: {Four=4, Two=2}

HashMap2: {Two=2, Three=3, Four=4}

## Methods of HashMap

The HashMap class provides various methods that allow us to perform various operations on the map.

## Insert Elements to HashMap

* put() - inserts the specified key/value mapping to the map
* putAll() - inserts all the entries from specified map to this map
* putIfAbsent() - inserts the specified key/value mapping to the map if the specified key is not present in the map

For example,

import java.util.HashMap;

class Main {

public static void main(String[] args) {

// Creating HashMap of even numbers

HashMap<String, Integer> evenNumbers = new HashMap<>();

// Using put()

evenNumbers.put("Two", 2);

evenNumbers.put("Four", 4);

// Using putIfAbsent()

evenNumbers.putIfAbsent("Six", 6);

System.out.println("HashMap of even numbers: " + evenNumbers);

//Creating HashMap of numbers

HashMap<String, Integer> numbers = new HashMap<>();

numbers.put("One", 1);

// Using putAll()

numbers.putAll(evenNumbers);

System.out.println("HashMap of numbers: " + numbers);

}

}

**Output**

HashMap of even numbers: {Six=6, Four=4, Two=2}

HashMap of numbers: {Six=6, One=1, Four=4, Two=2}

## Access HashMap Elements

**1. Using entrySet(), keySet() and values()**

* entrySet() - returns a set of all the key/value mapping of the map
* keySet() - returns a set of all the keys of the map
* values() - returns a set of all the values of the map

For example,

import java.util.HashMap;

class Main {

public static void main(String[] args) {

HashMap<String, Integer> numbers = new HashMap<>();

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("HashMap: " + numbers);

// Using entrySet()

System.out.println("Key/Value mappings: " + numbers.entrySet());

// Using keySet()

System.out.println("Keys: " + numbers.keySet());

// Using values()

System.out.println("Values: " + numbers.values());

}

}

**Output**

HashMap: {One=1, Two=2, Three=3}

Key/Value mappings: [One=1, Two=2, Three=3]

Keys: [One, Two, Three]

Values: [1, 2, 3]

**2. Using get() and getOrDefault()**

* get() - Returns the value associated with the specified key. Returns null if the key is not found.
* getOrDefault() - Returns the value associated with the specified key. Returns the specified default value if the key is not found.

For example,

import java.util.HashMap;

class Main {

public static void main(String[] args) {

HashMap<String, Integer> numbers = new HashMap<>();

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("HashMap: " + numbers);

// Using get()

int value1 = numbers.get("Three");

System.out.println("Returned Number: " + value1);

// Using getOrDefault()

int value2 = numbers.getOrDefault("Five", 5);

System.out.println("Returned Number: " + value2);

}

}

**Output**

HashMap: {One=1, Two=2, Three=3}

Returned Number: 3

Returned Number: 5

## Remove Elements

* remove(key) - returns and removes the entry associated with the specified key from the map
* remove(key, value) - removes the entry from the map only if the specified key mapped to the specified value and return a boolean value

For example,

import java.util.HashMap;

class Main {

public static void main(String[] args) {

HashMap<String, Integer> numbers = new HashMap<>();

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("HashMap: " + numbers);

// remove method with single parameter

int value = numbers.remove("Two");

System.out.println("Removed value: " + value);

// remove method with two parameters

boolean result = numbers.remove("Three", 3);

System.out.println("Is the entry Three removed? " + result);

System.out.println("Updated HashMap: " + numbers);

}

}

**Output**

HashMap: {One=1, Two=2, Three=3}

Removed value: 2

Is the entry Three removed? True

Updated HashMap: {One=1}

## Replace Elements

* replace(key, value) - replaces the value associated with the specified key by a new value
* replace(key, old, new) - replaces the old value with the new value only if old value is already associated with the specified key
* replaceAll(function) - replaces each value of the map with the result of the specified function

For example,

import java.util.HashMap;

class Main {

public static void main(String[] args) {

HashMap<String, Integer> numbers = new HashMap<>();

numbers.put("First", 1);

numbers.put("Second", 2);

numbers.put("Third", 3);

System.out.println("Original HashMap: " + numbers);

// Using replace()

numbers.replace("Second", 22);

numbers.replace("Third", 3, 33);

System.out.println("HashMap using replace(): " + numbers);

// Using replaceAll()

numbers.replaceAll((key, oldValue) -> oldValue + 2);

System.out.println("HashMap using replaceAll(): " + numbers);

}

}

**Output**

Original HashMap: {One=1, Two=2, Three=3}

HashMap using replace: {One=1, Two=22, Three=33}

HashMap using replaceAll: {One=2, Two=24, Three=35}

In the above program notice the statement

numbers.replaceAll((key, oldValue) -> oldValue + 2);

Here, the method accesses all the entries of the map. It then replaces all the values with the new values provided by the lambda expression.

## Recompute Values

**1. Using compute() Method**

* compute() - Computes a new value using the specified function. It then associates the computed value to the specified key.
* computeIfAbsent() - If the specified key is not mapped to any value, the method will compute a new value using the specified function. It then associates the new value with the key.
* computeIfPresent() - If the specified key is already mapped to any value, this method will compute a new value using the specified function. It then associates the new value with the key.

For example,

import java.util.HashMap;

class Main {

public static void main(String[] args) {

HashMap<String, Integer> numbers = new HashMap<>();

numbers.put("First", 1);

numbers.put("Second", 2);

System.out.println("Original HashMap: " + numbers);

// Using compute()

numbers.compute("First", (key, oldValue) -> oldValue + 2);

numbers.compute("Second", (key, oldValue) -> oldValue + 1);

System.out.println("HashMap using compute(): " + numbers);

// Using computeIfAbsent()

numbers.computeIfAbsent("Three", key -> 5);

System.out.println("HashMap using computeIfAbsent(): " + numbers);

// Using computeIfPresent()

numbers.computeIfPresent("Second", (key, oldValue) -> oldValue \* 2);

System.out.println("HashMap using computeIfPresent(): " + numbers);

}

}

**Output**

Original HashMap: {Second=2, First=1}

HashMap using compute(): {Second=5, First=3}

HashMap using computeIfAbsent(): {Second=5 First=3, Three=8}

HashMap using computeIfPresent(): {Second=10, First=3, three=8

In the above example, we have recomputed the values of the map using the compute() method.

Here, we have used lambda expressions as method arguments to recompute the values.

**2. Using merge() Method**

The merge() method associates the specified value to the specified key if the specified key is not already associated.

However, if the specified key is already associated with a value, it will merge the new specified value with the existing old value. For example,

import java.util.HashMap;

class Main {

public static void main(String[] args) {

HashMap<String, Integer> numbers = new HashMap<>();

numbers.put("First", 1);

numbers.put("Second", 2);

System.out.println("Original HashMap: " + numbers);

// Using merge() Method

numbers.merge("First", 4, (oldValue, newValue) -> oldValue + newValue);

System.out.println("New HashMap: " + numbers);

}

}

**Output**

Original HashMap: {Second=2, First=1}

New HashMap: {Second=2, First=5}

In the above example, the merge() method takes 3 parameters: **key**, **newValue** and a lambda expression (that computes the new merged value)

## Other Methods of HashMap

|  |  |
| --- | --- |
| Method | Description |
| clear() | Removes all the entries from the map |
| containsKey() | Checks if the map contains the specified key and returns a boolean value |
| containsValue() | Checks if the map contains the specified value and returns a boolean value |
| size() | Returns the size of the map |
| isEmpty() | Checks if the map is empty and returns a boolean value |

## Iterate Through a HashMap

In a HashMap, we can

* iterate through its **keys**
* iterate through its **values**
* iterate through its **keys/values**

**1. Using the forEach loop**

import java.util.HashMap;

import java.util.Map.Entry;

class Main {

public static void main(String[] args) {

// Creating a HashMap

HashMap<String, Integer> numbers = new HashMap<>();

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("HashMap: " + numbers);

// Accessing the key/value pair

System.out.print("Entries: ");

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

System.out.print(entry);

System.out.print(", ");

}

// Accessing the key

System.out.print("\nKeys: ");

for(String key: numbers.keySet()) {

System.out.print(key);

System.out.print(", ");

}

// Accessing the value

System.out.print("\nValues: ");

for(Integer value: numbers.values()) {

System.out.print(value);

System.out.print(", ");

}

}

}

**Output**

HashMap: {One=1, Two=2, Three=3}

Entries: One=1, Two=2, Three=3

Keys: One, Two, Three,

Values: 1, 2, ,3,

In the above program, note that we have imported the java.util.Map.Entry package. Here, Map.Entry is the nested class of the Map interface.

This nested class returns a view (elements) of the map.

**2. Using iterator() Method**

It is also possible to iterate a HashMap using the iterator() method. In order to use this method, we must import the java.util.Iterator package.

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map.Entry;

class Main {

public static void main(String[] args) {

// Creating a HashMap

HashMap<String, Integer> numbers = new HashMap<>();

numbers.put("One", 1);

numbers.put("Two", 2);

numbers.put("Three", 3);

System.out.println("HashMap: " + numbers);

// Creating an object of Iterator

Iterator<Entry<String, Integer>> iterate1 = numbers.entrySet().iterator();

// Accessing the Key/Value pair

System.out.print("Entries: ");

while(iterate1.hasNext()) {

System.out.print(iterate1.next());

System.out.print(", ");

}

// Accessing the key

Iterator<String> iterate2 = numbers.keySet().iterator();

System.out.print("\nKeys: ");

while(iterate2.hasNext()) {

System.out.print(iterate2.next());

System.out.print(", ");

}

// Accessing the value

Iterator<Integer> iterate3 = numbers.values().iterator();

System.out.print("\nValues: ");

while(iterate3.hasNext()) {

System.out.print(iterate3.next());

System.out.print(", ");

}

}

}

**Output**

HashMap: {One=1, Two=2, Three=3}

Entries: One=1, Two=2, Three=3

Keys: One, Two, Three,

Values: 1, 2, 3,

In the above program, note that we have imported the java.util.Map.Entry package. Here, Map.Entry is the nested class of the Map interface.

This nested class returns a view (elements) of the map.