

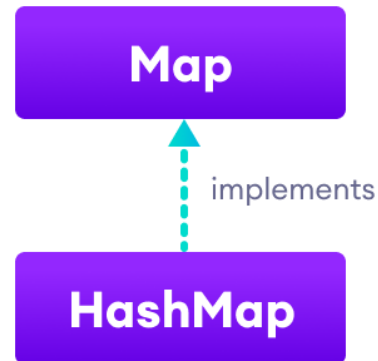
Java HashMap

In this tutorial, we will learn about the Java HashMap class and its various operations with the help of examples.

The `HashMap` class of the Java collections framework provides the functionality of the [hash table data structure](#).

It stores elements in **key/value** pairs. Here, **keys** are unique identifiers used to associate each **value** on a map.

The `HashMap` class implements the [Map](#) interface.



Java HashMap Implementation

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<K, V> numbers = new HashMap<>();
```

In the above code, we have created a hashmap named `numbers`. Here, **K** represents the key type and **V** represents the type of values. For example,

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

Here, the type of **keys** is `String` and the type of **values** is `Integer`.

Example 1: Create HashMap in Java

```
import java.util.HashMap;

class Main {
    public static void main(String[] args) {

        // create a hashmap
        HashMap<String, Integer> languages = new HashMap<>();

        // add elements to hashmap
        languages.put("Java", 8);
        languages.put("JavaScript", 1);
        languages.put("Python", 3);
        System.out.println("HashMap: " + languages);
    }
}
```

Output

```
HashMap: {Java=8, JavaScript=1, Python=3}
```

In the above example, we have created a `HashMap` named `languages`.

Here, we have used the `put()` method to add elements to the hashmap. We will learn more about the `put()` method later in this tutorial.

Basic Operations on Java HashMap

The `HashMap` class provides various methods to perform different operations on hashmaps. We will look at some commonly used arraylist operations in this tutorial:

- Add elements
 - Access elements
 - Change elements
 - Remove elements
-

1. Add elements to a HashMap

To add a single element to the hashmap, we use the `put()` method of the `HashMap` class. For example,

```
import java.util.HashMap;

class Main {
    public static void main(String[] args) {

        // create a hashmap
        HashMap<String, Integer> numbers = new HashMap<>();

        System.out.println("Initial HashMap: " + numbers);
        // put() method to add elements
        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("HashMap after put(): " + numbers);
    }
}
```

Output

```
Initial HashMap: {}
HashMap after put(): {One=1, Two=2, Three=3}
```

In the above example, we have created a `HashMap` named `numbers`. Here, we have used the `put()` method to add elements to numbers.

Notice the statement,

```
numbers.put("One", 1);
```

Here, we are passing the `String` value `One` as the key and `Integer` value `1` as the value to the `put()` method.

Recommended Readings

- [Java HashMap.put\(\)](#)
 - [Java HashMap.putAll\(\)](#)
 - [Java HashMap.putIfAbsent\(\)](#)
-

2. Access HashMap Elements

We can use the `get()` method to access the value from the hashmap. For example,

```
import java.util.HashMap;

class Main {
    public static void main(String[] args) {

        HashMap<Integer, String> languages = new HashMap<>();
        languages.put(1, "Java");
        languages.put(2, "Python");
        languages.put(3, "JavaScript");
        System.out.println("HashMap: " + languages);

        // get() method to get value
        String value = languages.get(1);
        System.out.println("Value at index 1: " + value);
    }
}
```

Output

```
HashMap: {1=Java, 2=Python, 3=JavaScript}
Value at index 1: Java
```

In the above example, notice the expression,

```
languages.get(1);
```

Here, the `get()` method takes the **key** as its argument and returns the corresponding **value** associated with the key.

We can also access the **keys**, **values**, and **key/value** pairs of the hashmap as set views using `keySet()`, `values()`, and `entrySet()` methods respectively. For example,

```
import java.util.HashMap;

class Main {
    public static void main(String[] args) {
        HashMap<Integer, String> languages = new HashMap<>();

        languages.put(1, "Java");
        languages.put(2, "Python");
        languages.put(3, "JavaScript");
        System.out.println("HashMap: " + languages);

        // return set view of keys
        // using keySet()
        System.out.println("Keys: " + languages.keySet());

        // return set view of values
        // using values()
        System.out.println("Values: " + languages.values());

        // return set view of key/value pairs
        // using entrySet()
        System.out.println("Key/Value mappings: " + languages.entrySet());
    }
}
```

Output


```
HashMap: {1=Java, 2=Python, 3=JavaScript}  
Keys: [1, 2, 3]  
Values: [Java, Python, JavaScript]  
Key/Value mappings: [1=Java, 2=Python, 3=JavaScript]
```

In the above example, we have created a hashmap named `languages`. Here, we are accessing the **keys**, **values**, and **key/value** mappings from the hashmap.

Recommended Readings

- [Java HashMap_get\(\)](#)
- [Java Hashmap_getOrDefault\(\)](#)
- [Java HashMap_keySet\(\)](#)

- [Java HashMap values\(\)](#)
 - [Java HashMap entrySet\(\)](#)
-

3. Change HashMap Value

We can use the `replace()` method to change the value associated with a key in a hashmap. For example,

```
import java.util.HashMap;

class Main {
    public static void main(String[] args) {

        HashMap<Integer, String> languages = new HashMap<>();
        languages.put(1, "Java");
        languages.put(2, "Python");
        languages.put(3, "JavaScript");
        System.out.println("Original HashMap: " + languages);

        // change element with key 2
        languages.replace(2, "C++");
        System.out.println("HashMap using replace(): " + languages);
    }
}
```

Output

```
Original HashMap: {1=Java, 2=Python, 3=JavaScript}  
HashMap using replace(): {1=Java, 2=C++, 3=JavaScript}
```

In the above example, we have created a hashmap named `languages`. Notice the expression,

```
languages.replace(2, "C++");
```

Here, we are changing the value referred to by key **2** with the new value `C++`.

The `HashMap` class also provides some variations of the `replace()` method. To learn more, visit

- [Java HashMap replace\(\)](#)
- [Java HashMap replaceAll\(\)](#)

4. Remove HashMap Elements

To remove elements from a hashmap, we can use the `remove()` method. For example,

```
import java.util.HashMap;

class Main {
    public static void main(String[] args) {

        HashMap<Integer, String> languages = new HashMap<>();
        languages.put(1, "Java");
        languages.put(2, "Python");
        languages.put(3, "JavaScript");
        System.out.println("HashMap: " + languages);

        // remove element associated with key 2
        String value = languages.remove(2);
        System.out.println("Removed value: " + value);

        System.out.println("Updated HashMap: " + languages);
    }
}
```

Output

```
HashMap: {1=Java, 2=Python, 3=JavaScript}
Removed value: Python
Updated HashMap: {1=Java, 3=JavaScript}
```

Here, the `remove()` method takes the **key** as its parameter. It then returns the **value** associated with the **key** and removes the **entry**.

We can also remove the entry only under certain conditions. For example,

```
remove(2, "C++");
```

Here, the `remove()` method only removes the **entry** if the **key 2** is associated with the **value C++**. Since **2** is not associated with **C++**, it doesn't remove the entry.

To learn more, visit [Java HashMap remove\(\)](#).

Other Methods of HashMap

Method	Description
clear()	removes all mappings from the <code>HashMap</code>
compute()	computes a new value for the specified key
computeIfAbsent()	computes value if a mapping for the key is not present
computeIfPresent()	computes a value for mapping if the key is present
merge()	merges the specified mapping to the <code>HashMap</code>
clone()	makes the copy of the <code>HashMap</code>
containsKey()	checks if the specified key is present in Hashmap
containsValue()	checks if <code>Hashmap</code> contains the specified value

[size\(\)](#)

returns the number of items in `HashMap`

[isEmpty\(\)](#)

checks if the `Hashmap` is empty

Iterate through a HashMap

To iterate through each entry of the hashmap, we can use [Java for-each loop](#). We can iterate through **keys only**, **vales only**, and **key/value mapping**. For example,

```
import java.util.HashMap;
import java.util.Map.Entry;

class Main {
    public static void main(String[] args) {

        // create a HashMap
        HashMap<Integer, String> languages = new HashMap<>();
        languages.put(1, "Java");
        languages.put(2, "Python");
        languages.put(3, "JavaScript");
        System.out.println("HashMap: " + languages);

        // iterate through keys only
        System.out.print("Keys: ");
        for (Integer key : languages.keySet()) {
            System.out.print(key);
            System.out.print(", ");
        }

        // iterate through values only
        System.out.print("\nValues: ");
        for (String value : languages.values()) {
            System.out.print(value);
            System.out.print(", ");
        }

        // iterate through key/value entries
```

```
System.out.print("\nEntries: ");
```

Output

```
HashMap: {1=Java, 2=Python, 3=JavaScript}  
Keys: 1, 2, 3,  
Values: Java, Python, JavaScript,  
Entries: 1=Java, 2=Python, 3=JavaScript,
```

Note that we have used the `Map.Entry` in the above example. It is the nested class of the `Map` interface that returns a view (elements) of the map.

We first need to import the `java.util.Map.Entry` package in order to use this class.

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

Creating HashMap from Other Maps

In Java, we can also create a hashmap from other maps. For example,


```
import java.util.HashMap;
import java.util.TreeMap;

class Main {
    public static void main(String[] args) {

        // create a treemap
        TreeMap<String, Integer> evenNumbers = new TreeMap<>();
        evenNumbers.put("Two", 2);
        evenNumbers.put("Four", 4);
        System.out.println("TreeMap: " + evenNumbers);

        // create hashmap from the treemap
        HashMap<String, Integer> numbers = new HashMap<>(evenNumbers);
        numbers.put("Three", 3);
        System.out.println("HashMap: " + numbers);
    }
}
```

Output

```
TreeMap: {Four=4, Two=2}
HashMap: {Two=2, Three=3, Four=4}
```

In the above example, we have created a `TreeMap` named `evenNumbers`. Notice the expression,

```
numbers = new HashMap<>(evenNumbers)
```

Here, we are creating a `HashMap` named `numbers` using the `TreeMap`. To learn more about treemap, visit [Java TreeMap](#).

Note: While creating a hashmap, we can include optional parameters: **capacity** and **load factor**. For example,

```
HashMap<K, V> numbers = new HashMap<>(8, 0.6f);
```

Here,

- **8** (capacity is 8) - This means it can store 8 entries.
- **0.6f** (load factor is 0.6) - This means whenever our hash table is filled by 60%, the entries are moved to a new hash table double the size of the original hash table.

If the optional parameters not used, then the default **capacity** will be **16** and the default **load factor** will be **0.75**.