HowToDoInJava

Java LinkedHashMap class

Last Updated: December 26, 2020 By: Lokesh Gupta Java Collections Java Map

LinkedHashMap in Java is used to store key-value pairs very similar to `HashMap` class. Difference is that LinkedHashMap maintains the order of elements inserted into it while HashMap is unordered.

In this Java collection tutorial, we will learn about LinkedHashMap class, it's methods, usecases and other important details.

Table of Contents

1. LinkedHashMap Hierarchy
2. LinkedHashMap Features
3. LinkedHashMap Constructors
4. LinkedHashMap Methods
5. LinkedHashMap Usecases
6. LinkedHashMap Performance
7. Concurrency in LinkedHashMap
8. Conclusion

1. LinkedHashMap Hierarchy

The LinkedHashMap class is declared as following in Java. It **extends HashMap** class and **implements Map** interface. Here `'K'` is the type of keys and `'V'` is the type of mapped values to keys.

```
LinkedHashMap.java

public class LinkedHashMap<K,V>
    extends HashMap<K,V>
    implements Map<K,V>
{
  //implementation
}
```

2. LinkedHashMap Features

The important things to learn about Java LinkedHashMap class are:

- It stores key-value pairs similar to HashMap.
- It contains only unique keys. Duplicate keys are not allowed.
- It may have one `null` key and multiple `null` values.
- It maintains the order of K,V pairs inserted to it by adding elements to internally managed **doubly-linked list**.

2.1. Insertion ordered LinkedHashMap

By default, LinkedHashMap is insertion ordered. It maintains the order of elements when they were added to it. While iterating over LinkedHashMap, we get the KV pairs in exact order they were added.

```
Insertion ordered LinkedHashMap Example

LinkedHashMap<Integer, String> pairs = new LinkedHashMap<>();

pairs.put(1,  "A");
pairs.put(2,  "B");
pairs.put(3,  "C");
pairs.put(4,  "D");

pairs.forEach((key, value) -> {
    System.out.println("Key:"+ key + ", Value:" + value);
});
```

Program Output.

```
Console

Key:1, Value:A
Key:2, Value:B
Key:3, Value:C
Key:4, Value:D
```

2.2. Access ordered LinkedHashMap

In access ordered map, keys are sorted on the basis of access order last time they were accessed using any method of LinkedHashMap. Invoking the put, putIfAbsent, get, getOrDefault, compute, computeIfAbsent, computeIfPresent, or merge methods results in an access to the corresponding entry.

The keys are sorted from least recently accessed used to most recently accessed and build a LRU cache.

To create access order map, LinkedHashMap has a special constructor argument. When set to `true`, LinkedHashMap maintains the access order.

```
Access ordered LinkedHashMap Example

//3rd parameter set access order
LinkedHashMap<Integer, String> pairs = new LinkedHashMap<>(2, .75f, true);

pairs.put(1,  "A");
pairs.put(2,  "B");
pairs.put(3,  "C");
pairs.put(4,  "D");

//Access 3rd pair
pairs.get(3);

//Access 1st pair
pairs.getOrDefault(2, "oops");

pairs.forEach((key, value) -> {
    System.out.println("Key:"+ key + ", Value:" + value);
});
```

Program Output.

```
Console

Key:1, Value:A
Key:4, Value:D
Key:3, Value:C
Key:2, Value:B
```

Notice the output that how most recently accessed entry goes to the end of order.

3. LinkedHashMap Constructors

The LinkedHashMap has five types of constructors:

1. **LinkedHashMap():** initializes a default LinkedHashMap implementation with the default initial capacity (16) and load factor (0.75).
2. **LinkedHashMap(int capacity):** initializes a LinkedHashMap with a specified capacity and load factor (0.75).
3. **LinkedHashMap(Map map):** initializes a LinkedHashMap with same mappings as the specified map.
4. **LinkedHashMap(int capacity, float fillRatio):** initializes LinkedHashMap with specified initial capacity and load factor.
5. **LinkedHashMap(int capacity, float fillRatio, boolean Order):** initializes both the capacity and fill ratio for a LinkedHashMap along with whether to maintain the insertion order or access order.
   - `'true'` enable access order.
   - `'false'` enable insertion order. This is default value behavior when using other constructors.

4. LinkedHashMap Methods

The important methods we should learn about LinkedHashMap are as follows:

1. **void clear():** It removes all the key-value pairs from the map.

2. **void size():** It returns the number of key-value pairs present in this map.
3. **void isEmpty():** It returns true if this map contains no key-value mappings..
4. **boolean containsKey(Object key):** It returns `'true'` if a specified key is present in the map.
5. **boolean containsValue(Object key):** It returns `'true'` if a specified value is mapped to at least one key in the map.
6. **Object get(Object key):** It retrieves the `value` mapped by the specified `key`.
7. **Object remove(Object key):** It removes the key-value pair for the specified key from the map if present.
8. **boolean removeEldestEntry(Map.Entry eldest):** It returns `'true'` when the map removes its eldest entry from the access ordered map.

4.1. Java LinkedHashMap Example

Java program to demonstrate the usages of linkedhashmap methods.

```
LinkedHashMap examples

import java.util.Iterator;
import java.util.LinkedHashMap;

public class LinkedHashMapExample
{
    public static void main(String[] args)
    {
        //3rd parameter set access order
        LinkedHashMap<Integer, String> pairs = new LinkedHashMap<>();

        pairs.put(1,  "A");
        pairs.put(2,  "B");
        pairs.put(3,  "C");

        String value = pairs.get(3);    //get method

        System.out.println(value);

        value = pairs.getOrDefault(5, "oops");  //getOrDefault method

        System.out.println(value);
```

```
        //Iteration example
        Iterator<Integer> iterator =  pairs.keySet().iterator();

        while(iterator.hasNext()) {
            Integer key = iterator.next();
            System.out.println("Key: " + key + ", Value: " + pairs.get(key));
        }

        //Remove example
        pairs.remove(3);
        System.out.println(pairs);

        System.out.println(pairs.containsKey(1));    //containsKey method

        System.out.println(pairs.containsValue("B"));    //containsValue method
    }
}
```

Program Output.

```
Console

C
oops
Key: 1, Value: A
Key: 2, Value: B
Key: 3, Value: C
{1=A, 2=B}
true
true
```

5. LinkedHashMap Usecases

We can use LinkedHashMap in almost all situations where we require to use HashMap. Functionlity wise it can replace HashMap very transparentely.

Additionally, LinkedHashMap maintains the insertion order which makes it super useful when we want to maintain the order of pairs added to the Map.

Access ordered LinkedHashMap provides a great starting point for creating a **LRU Cache** functionality by overriding the `removeEldestEntry()` method to impose a

policy for automatically removing stale when new mappings are added to the map. This lets you expire data using some criteria that you define.

6. LinkedHashMap Performance

HashMap and LinkedHashMap performs the basic operations of add, remove and contains in constant-time performance. LinkedHashMap performs a little wose than HashMap because it has to maintain a doubly-linkedlist and HashMap maintain only linked list.

On the other hand, looping over Map in the case of LinkedHashMap is slightly faster than HashMap because the time required is proportional to 'size' only. In case of HashMap, iteration performance of proportional to 'size + capacity'.

7. Concurrency in LinkedHashMap

Both HashMap and LinkedHashMap are **not thread-safe** which means we can not directly use them in a multi-threaded application for consistent results. We should synchronize them explicitely by using **Collections.synchronizedMap(Map map)** method.

```
Collections.synchronizedMap() usage

Map<Integer, Integer> numbers = Collections.synchronizedMap(new LinkedHashMap<>())

Map<Integer, Integer> numbers = Collections.synchronizedMap(new HashMap<>());
```

In case of HashMap, use of **ConcurrentHashMap** is more advisable because of much higher degree of concurrency it provides.

8. Conclusion

Based on all above information, we can say that it is always better to choose HashMap over LinkedHashMap in most of the scenarios. We can prefer LinkedHashMap only when we have certain requirement or usecase which requires to maintain the order of elements added to the map.

Both provide pretty much same performance in most of the real world usecases. When we have a very large volume of data, then only we should be cosidering teh trade offs between them.

Happy Learning !!

Reference:

LinkedHashMap Java Docs

Recommended Reading:

1. [Solved]: javax.xml.bind.JAXBException: class java.util.ArrayList nor any of its super class is known to this context
2. Java TreeMap class
3. Java TransferQueue – Java LinkedTransferQueue class
4. Java HashSet class
5. Java LinkedHashSet class

6. Java TreeSet class
7. Java LinkedList class
8. Java PriorityBlockingQueue class
9. Java CopyOnWriteArrayList class
0. Java CopyOnWriteArraySet class

Leave a Comment

Name *

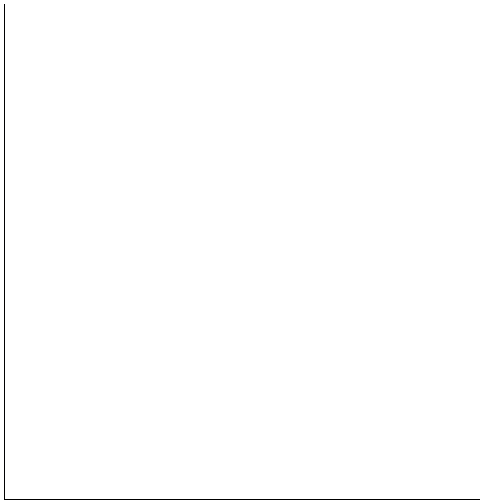
Email *

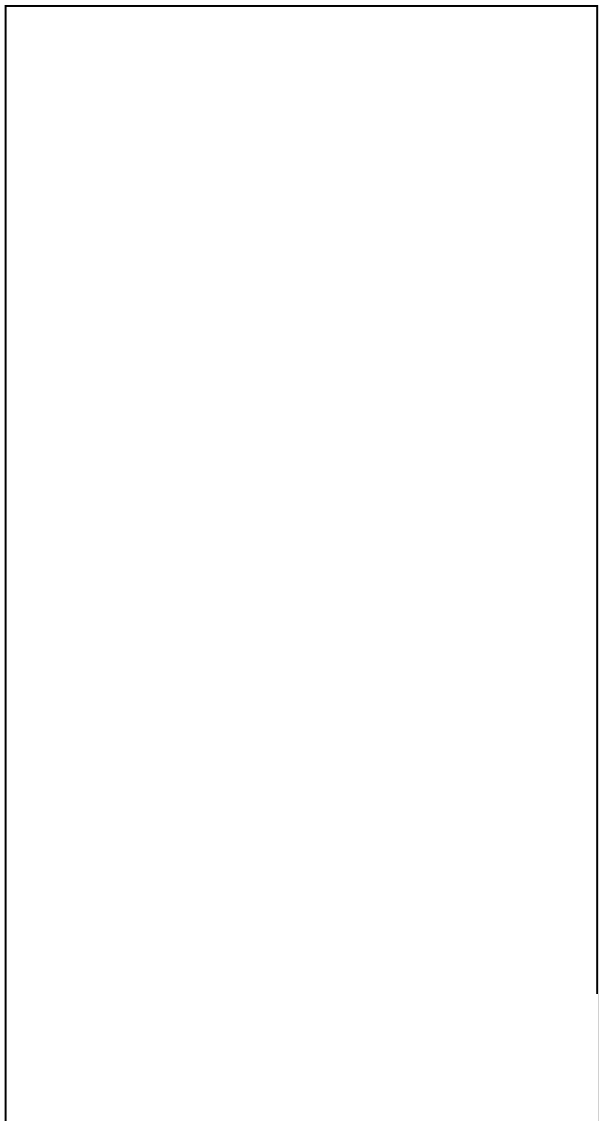
Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...





HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial