default V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)

Predefined method of Map interface from JDK 1.8V

It is used to compute and insert the value in the Map by using given Function<K,V>, If key is not available in the Map collection.

Important Points related to computeIfAbsent() :

Key already has a value → Do nothing, return that existing value.

Key missing or null → Create a new value (using the function) and put it in the map.

If the function gives null → Don't put anything.

```
package com.ravi.hash_map;

import java.util.HashMap;
import java.util.Map;

//V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)

public class HashMapDemo7
{
    public static void main(String[] args)
    {
       Map<String,Integer> map = new HashMap<>();

       map.computeIfAbsent("A", key -> key.length());
       map.computeIfAbsent("A", key -> 100);
       map.computeIfAbsent("B", key -> 200);
       map.computeIfAbsent("C", key -> null);
       map.computeIfAbsent(null, key -> 400);

       System.out.println(map);

    }
}
```

---

default V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)

It adds the entry in the map, if the key is new, otherwise it combines old and new key value with the given function. If function result is null then key will be removed.It is accept entrire Map and perform operation based on the given key.

Different Cases :
----
a) If the key is missing OR has null [Insert the given value]
map.merge("A", 10, (oldVal, newVal) -> oldVal + newVal);
"A" not present in the map → map = {A=10}

b) If the key already has a value
[Combine old value and new value using the function]
map.merge("A", 5, (oldVal, newVal) -> oldVal + newVal);
"A" was 10 → 10 + 5= 15 → map = {A=15}

c) If the function gives null [Remove the key from the map]
map.merge("A", 5, (oldVal, newVal) -> null);
Removes "A" from the Map.

```
package com.ravi.hash_map;

import java.util.HashMap;
import java.util.Map;

/*default V merge
     (K key, V value, BiFunction<? super V, ? super V, ? extends V>
remappingFunction)  */

public class HashMapDemo8
{
public static void main(String[] args)
{
        Map<String, Integer> map = new HashMap<>();

        map.merge("A",10, (oldValue, newValue)-> oldValue + newValue);
        map.merge("A",15, (oldValue, newValue)-> oldValue + newValue);
        map.merge("A",25, (oldValue, newValue)-> oldValue + newValue);
        map.merge("B",25, (oldValue, newValue)-> oldValue + newValue);
        map.merge("B",25, (oldValue, newValue)-> oldValue + newValue);

        System.out.println(map);

        System.out.println("=============================");


       Map<String,Integer> map1 = new HashMap<>();

       map1.computeIfAbsent("A", key -> key.length());
       map1.computeIfAbsent("B", key -> 10);
       map1.computeIfAbsent("A", key -> 20);
       map1.computeIfAbsent("C", key -> 30);

      map1.merge("B", 25, (oldValue, newValue)->  oldValue + newValue);
      System.out.println(map1);

      System.out.println("=============================");

      map1.merge("C", 30, (oldValue, newValue) -> null);
      System.out.println(map1);

    }
}
```

Map with List of values [Map<K, List<V>>]
----
default V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)

It changes the value only if the key is already there. If result is null, the key is removed from the Map.

Different Cases :
----
a) If the key exists in the map and its value is not null [Update the value using the function]

b) If the key is missing OR value is null [Do nothing]

c) If the function returns null [Remove the key]

```
package com.ravi.hash_map;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;


public class HashMapDemo9
{
public static void main(String[] args)
  {
        Map<String, List<String>> studentSubject = new HashMap<>();
        studentSubject.computeIfAbsent("Scott", key -> new ArrayList<String>
()).add("Phy");
        studentSubject.computeIfAbsent("Scott", key -> new ArrayList<String>
()).add("Che");

        studentSubject.computeIfAbsent("Alen", key -> new ArrayList<String>
()).add("History");
        studentSubject.computeIfAbsent("Alen", key -> new ArrayList<String>
()).add("Geography");

        System.out.println(studentSubject);


        studentSubject.computeIfPresent("Scott", (key, value)-> { value.add("Math");
return value;});

        studentSubject.computeIfPresent("Alen", (key, value)-> {
value.add("Economics"); return value;});

        System.out.println(studentSubject);

    }
}
```

```
package com.ravi.hash_map;

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

//Cheapest and expensive Flight


public class HashMapDemo10
{
   public static void main(String[] args)
   {
       // Map<DestinationCity, Price>

       Map<String, Integer> flights = new HashMap<>();
       flights.put("Mumbai", 5500);
       flights.put("Bangalore", 7000);
       flights.put("Chennai", 6500);
       flights.put("Vizag", 5000);
       flights.put("Pune", 5000);
       flights.put("Goa", 8000);
       flights.put("Kolkata", 8000);


       //Cheapest Flight
       Entry<String, Integer> min = Collections.min(flights.entrySet(), (f1, f2)->
f1.getValue().compareTo(f2.getValue()));
       System.out.println(min.getKey()+" : "+min.getValue());


       //Expensive Flight

       int max = Collections.max(flights.values());

       for(Map.Entry<String, Integer> entry : flights.entrySet())
       {
            if(entry.getValue() == max)
            {
               System.out.println("Expensive flight from Hyderabad to
"+entry.getKey()+" price is :"+entry.getValue());
            }
       }
    }
}
```

LinkedHashSet<E> [It is an ordered version of HashSet]
----
public class LinkedHashSet extends HashSet implements Set, Clonable, Serializable

It is a predefined class in java.util package under Set interface and introduced from java 1.4v.

It is the sub class of HashSet class.

It is an orderd version of HashSet that maintains a doubly linked list across all the elements.

It internally uses Hashtable and LinkedList data structures.

We should use LinkedHashSet class when we want to maintain an order.

When we iterate the elements through HashSet the order will be unpredictable, while when we iterate the elements through LinkedHashSet then the order will be same as they were inserted in the collection.

```
//Program :
import java.util.*;
public class LinkedHashSetDemo
{
 public static void main(String args[])
   {
        LinkedHashSet<String> lhs = new LinkedHashSet<>();
        lhs.add("Ravi");
        lhs.add("Vijay");
        lhs.add("Ravi");
        lhs.add("Ajay");
        lhs.add("Pawan");
        lhs.add("Shiva");
        lhs.add(null);
        lhs.add("Ganesh");
        lhs.forEach(str -> System.out.println(str));
   }
}
```

```
 import java.util.*;

 public class LinkedHashSetDemo1
 {
    public static void main(String[] args)
     {
      LinkedHashSet<Integer> linkedHashSet = new LinkedHashSet<>();

       linkedHashSet.add(10);
       linkedHashSet.add(5);
       linkedHashSet.add(15);
       linkedHashSet.add(20);
       linkedHashSet.add(5);

       System.out.println("LinkedHashSet elements: " + linkedHashSet);

       System.out.println("LinkedHashSet size: " + linkedHashSet.size());

       int elementToCheck = 15;
       if (linkedHashSet.contains(elementToCheck))
          {
           System.out.println(elementToCheck + " is present in the LinkedHashSet.");
       }
          else
          {
           System.out.println(elementToCheck + " is not present in the LinkedHashSet.");
       }

       int elementToRemove = 10;
       linkedHashSet.remove(elementToRemove);
       System.out.println("After removing " + elementToRemove + ", LinkedHashSet elements: " +
 linkedHashSet);

            linkedHashSet.clear();
       System.out.println("After clearing, LinkedHashSet elements: " + linkedHashSet); //[]
    }
 }
```

LinkedHashMap<K,V> [It maintains insertion order]
----
public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>

It is a predefined class available in java.util package under Map interface available from 1.4.

It is the sub class of HashMap class.

It maintains insertion order. It contains a doubly linked with the elements or nodes so It will iterate more slowly in comparison to HashMap.

It uses Hashtable and LinkedList data structure.

If We want to fetch the elements in the same order as they were inserted in the Map then we should go with LinkedHashMap.

It accepts one null key and multiple null values.

It is not synchronized.

It has also 4 constructors same as HashMap

1) LinkedHashMap hm1 = new LinkedHashMap();
will create a LinkedHashMap with default capacity 16 and load factor 0.75

2) LinkedHashMap hm1 = new LinkedHashMap(iny initialCapacity);

3) LinkedHashMap hm1 = new LinkedHashMap(int initialCapacity, float loadFactor);

4) LinkedHashMap hm1 = new LinkedHashMap(Map m);

```
import java.util.*;
public class LinkedHashMapDemo
{
    public static void main(String[] args)
    {
        LinkedHashMap<Integer,String> l = new LinkedHashMap<>();
        l.put(1,"abc");
        l.put(3,"xyz");
        l.put(2,"pqr");
        l.put(4,"def");
        l.put(null,"ghi");
        System.out.println(l);
    }
}
```

```
import java.util.*;

public class LinkedHashMapDemo1
{
    public static void main(String[] a)
    {
        Map<String,String> map = new LinkedHashMap<>();
        map.put("Ravi", "1234");
           map.put("Rahul", "1234");
           map.put("Aswin", null);
           map.put("Samir", null);

           map.forEach((k,v)->System.out.println(k+" : "+v));
    }
}
```