

Day 8- Facilitation Guide

Index

- I. Set interface implementation classes - LinkedHashSet, TreeSet
- II. Map interface introduction & implementation classes - HashMap, LinkedHashMap

For (1.5 hrs) ILT

During the previous session about Stack, Queue and the Set interface in Java, we covered a variety of important concepts:

We have learnt that Stack follows LIFO (Last-In-First-Out) and Queue follows the FIFO (First-In-First-Out) order. We also discussed the below characteristics of the Set interface.

No Duplicate Elements: A Set is a collection that does not allow duplicate elements. Each element in a set is unique.

Interface Hierarchy: The Set interface is part of the Java Collections Framework and extends the Collection interface. It does not introduce any new methods but inherits the methods from the Collection interface.

Subinterfaces: There are three main subinterfaces of Set in Java: HashSet, LinkedHashSet, and TreeSet. Each subinterface provides a different implementation for storing and managing the elements.

No Guaranteed Order: The Set interface does not guarantee any specific order of elements. The order of elements may be arbitrary and can change over time.

HashSet: HashSet is one of the most commonly used implementations of the Set interface. It uses a hash table to store elements and provides constant-time performance for basic operations like add, remove, and contains.

Null Elements: Most implementations of the Set interface do not allow null elements. However, HashSet permits a single null element, and the TreeSet implementation throws a NullPointerException for null elements.

Remember that the choice of which Set implementation to use depends on the specific requirements of your application, such as whether you need ordered or sorted elements, or if you have any constraints on element uniqueness.

In this session, we will explore LinkedHashSet and TreeSet and the Map interface along with its diverse implementation classes.

Food for thought:

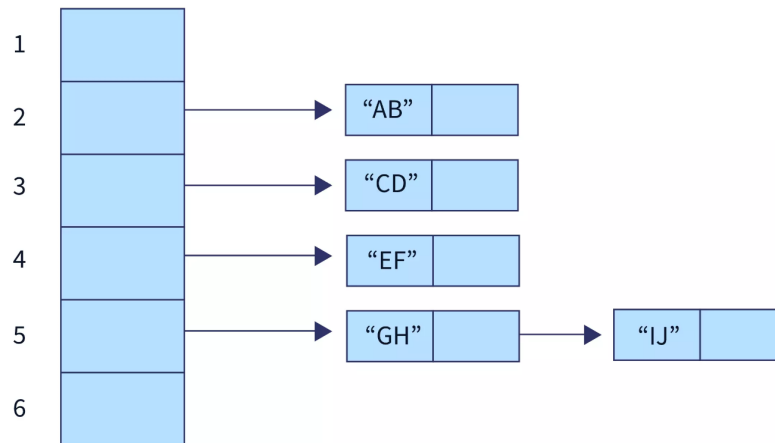
How does a LinkedHashSet maintain both insertion order and uniqueness of elements simultaneously?

LinkedHashSet

- The LinkedHashSet class extends the HashSet class.
- It maintains a linked list of entries in the set and hence maintains the order in which they were inserted.
- LinkedHashSet is non-synchronized, meaning multiple threads at a time can access the code. This means if one thread is working on LinkedHashSet, other threads can also get a hold of it. Multiple operations on LinkedHashSet can be performed at a time. For example, if addition is being performed by one thread, other operations can be performed by some other thread too.

Note: We are going to discuss Thread in the upcoming sessions.

The figure below shows the illustration of a LinkedHashSet:



Sr.No	Constructor & Description
1	HashSet() This constructor constructs a default HashSet.
2	HashSet(Collection c) This constructor initializes the hash set by using the elements of the collection c.
3	LinkedHashSet(int capacity) This constructor initializes the capacity of the LinkedHashSet to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
4	LinkedHashSet(int capacity, float fillRatio) This constructor initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments.

Example:

```

package com.anudip;
import java.util.LinkedHashSet;
public class LinkedHashSetDemo {
    public static void main(String[] args) {
        // create a hash set
  
```

```
LinkedHashSet<String> hs = new LinkedHashSet<String>();
```

```
// add elements to the hash set
```

```
hs.add("B");
```

```
hs.add("A");
```

```
hs.add("D");
```

```
hs.add("E");
```

```
hs.add("C");
```

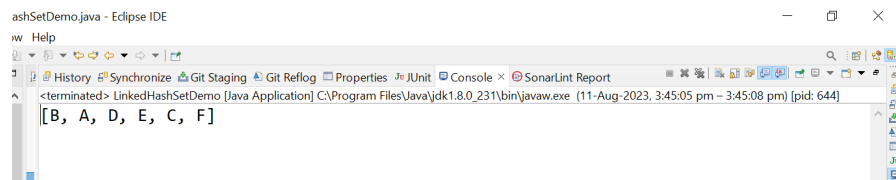
```
hs.add("F");
```

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

```
}
```

```
}
```

Output



*Let's understand the purpose of using a **LinkedHashSet** over a regular **HashSet**:*

The main purpose of using a **LinkedHashSet** over a regular **HashSet** in Java is to maintain the order of insertion while still benefiting from the features of a hash-based set. Here are some specific advantages of using a **LinkedHashSet** over a **HashSet**:

Order Preservation: The key advantage of a **LinkedHashSet** is that it preserves the order of insertion. When you iterate through a **LinkedHashSet**, the elements are returned in the same order they were added. This can be useful in scenarios where the order of elements matters, such as maintaining a history of operations or processing data in a specific sequence.

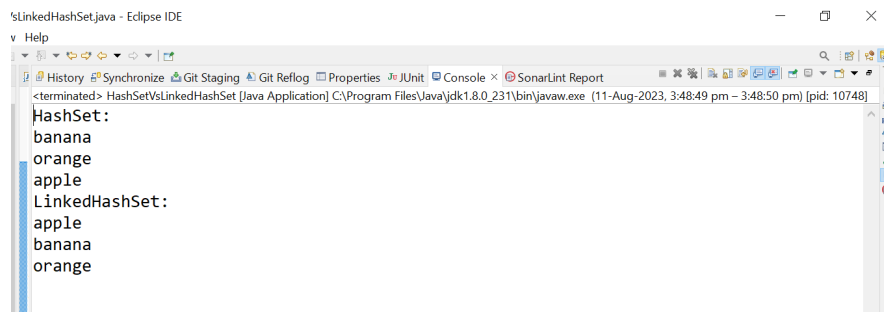
Removing Duplicates and Preserving Order: If you want to eliminate duplicate elements from a collection while retaining the order of the remaining elements, a **LinkedHashSet** is a natural fit. It ensures that you have a collection with distinct elements while still maintaining their insertion order.

Maintaining Consistency: If you are building data structures or algorithms that depend on the order of elements, a **LinkedHashSet** can ensure consistent behavior, whereas a regular **HashSet** might reorder elements as it rehashes and resizes.

Let's see the example

```
package com.anudip;
import java.util.HashSet;
import java.util.LinkedHashSet;
public class HashSetVsLinkedHashSet {
    public static void main(String[] args) {
        HashSet<String> hashSet = new HashSet<>();
        LinkedHashSet<String> linkedHashSet = new LinkedHashSet<>();
        hashSet.add("apple");
        hashSet.add("banana");
        hashSet.add("orange");
        linkedHashSet.add("apple");
        linkedHashSet.add("banana");
        linkedHashSet.add("orange");
        System.out.println("HashSet:");
        for (String fruit : hashSet) {
            System.out.println(fruit);
        }
        System.out.println("LinkedHashSet:");
        for (String fruit : linkedHashSet) {
            System.out.println(fruit);
        }
    }
}
```

Output:



In this example, the HashSet doesn't guarantee any specific order when iterating, while the LinkedHashSet maintains the order of insertion.

TreeSet Class

TreeSet is similar to HashSet except that it sorts the elements in the ascending order while HashSet doesn't maintain any order. TreeSet allows null elements but like HashSet it doesn't allow. Like most of the other collection classes this class is also not synchronized, however it can be synchronized explicitly like this: `SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));`

Here are some key features and benefits of using a TreeSet:

Sorted Order: The primary purpose of a TreeSet is to maintain a sorted order of elements. Elements are automatically arranged based on their natural order or a custom order specified through a Comparator.

Efficient Insertion and Retrieval: The Red-Black Tree structure ensures that basic operations like insertion, removal, and retrieval of elements have a time complexity of $O(\log n)$, where n is the number of elements in the Set. This makes TreeSet a good choice when you need sorted data and efficient access.

No Duplicates: Like other Set implementations, a TreeSet ensures that duplicate elements are not allowed. This property can be useful when you want to maintain a collection of unique items in a sorted manner.

Custom Ordering: You can provide a custom Comparator to define a specific sorting order for the elements in the TreeSet. This is useful when the natural ordering of elements doesn't meet your requirements.

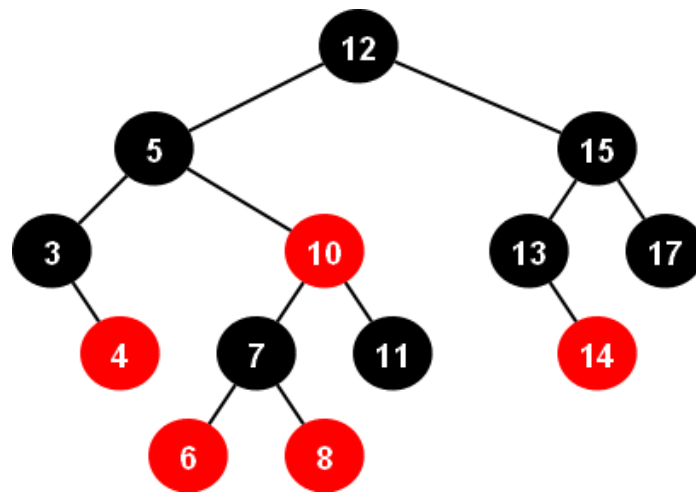
Navigable Set Operations: The TreeSet class extends the NavigableSet interface, providing additional methods for navigating the Set based on elements. You can perform operations like finding the nearest element, getting subsets of elements, and more.

Useful for Range Queries: The sorted nature of a TreeSet makes it suitable for range queries. You can easily find elements within a specified range or perform range-based operations.

Question: *What is the role of Red-Black Trees in TreeSet? How does a Red-Black Tree balancing property help with maintaining efficient operations?*

Ans- The role of Red-Black Trees in a TreeSet is to provide a balanced and efficient way to maintain the sorted order of elements. Red-Black Trees are a type of self-balancing binary search tree, and their balancing property ensures that the tree

remains relatively balanced even after multiple insertions and deletions. This balanced structure directly contributes to maintaining efficient operations in TreeSet.



- The root is black
- The children of a red node are black
- Every path from the root to a 0-node or a 1-node has the same number of black nodes

Sr.No.	Constructor & Description
1	<code>TreeSet()</code> This constructor constructs an empty tree set that will be sorted in an ascending order according to the natural order of its elements.
2	<code>TreeSet(Collection c)</code> This constructor builds a tree set that contains the elements of the collection c.
3	<code>TreeSet(Comparator comp)</code> This constructor constructs an empty tree set that will be sorted according to the given comparator.

4	TreeSet(SortedSet ss) This constructor builds a TreeSet that contains the elements of the given SortedSet.
---	--

Apart from the methods inherited from its parent classes, TreeSet defines the following methods –

Sr.No	Method & Description
1	void add(Object o) Adds the specified element to this set if it is not already present.
2	boolean addAll(Collection c) Adds all of the elements in the specified collection to this set.
3	void clear() Removes all of the elements from this set.
4	Object clone() Returns a shallow copy of this TreeSet instance.
5	Comparator comparator() Returns the comparator used to order this sorted set, or null if this tree set uses its elements natural ordering.

6	<p><code>boolean contains(Object o)</code></p> <p>Returns true if this set contains the specified element.</p>
7	<p><code>Object first()</code></p> <p>Returns the first (lowest) element currently in this sorted set.</p>
8	<p><code>SortedSet headSet(Object toElement)</code></p> <p>Returns a view of the portion of this set whose elements are strictly less than toElement.</p>
9	<p><code>boolean isEmpty()</code></p> <p>Returns true if this set contains no elements.</p>
10	<p><code>Iterator iterator()</code></p> <p>Returns an iterator over the elements in this set.</p>
11	<p><code>Object last()</code></p> <p>Returns the last (highest) element currently in this sorted set.</p>
12	<p><code>boolean remove(Object o)</code></p> <p>Removes the specified element from this set if it is present.</p>
13	<p><code>int size()</code></p> <p>Returns the number of elements in this set (its cardinality).</p>

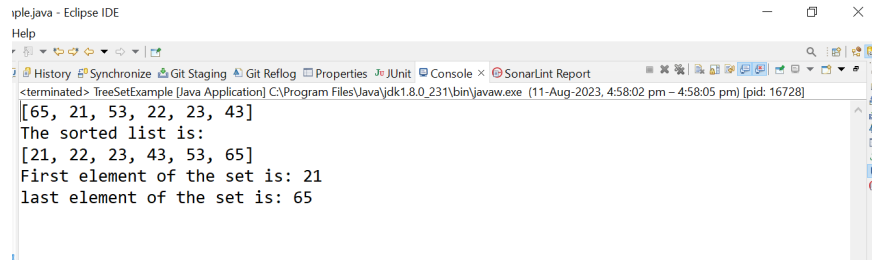
14	<p>SortedSet subSet(Object fromElement, Object toElement)</p> <p>Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.</p>
15	<p>SortedSet tailSet(Object fromElement)</p> <p>Returns a view of the portion of this set whose elements are greater than or equal to fromElement.</p>

Example:

```
package com.anudip;
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;
public class TreeSetExample {
    public static void main(String[] args) {
        int count[] = { 21, 23, 43, 53, 22, 65 };
        Set<Integer> set = new HashSet<Integer>();
        try {
            for (int i = 0; i <= 5; i++) {
                set.add(count[i]);
            }
            System.out.println(set);

            TreeSet<Integer> sortedSet = new TreeSet<Integer>(set);
            System.out.println("The sorted list is:");
            System.out.println(sortedSet);
            System.out.println("First element of the set is: " + (Integer) sortedSet.first());
            System.out.println("last element of the set is: " + (Integer) sortedSet.last());
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Output:

A screenshot of the Eclipse IDE's console window. The title bar reads 'iple.java - Eclipse IDE'. The console shows the following output:

```
[65, 21, 53, 22, 23, 43]
The sorted list is:
[21, 22, 23, 43, 53, 65]
First element of the set is: 21
last element of the set is: 65
```

The console also shows a status bar at the bottom with various icons and a message: '<terminated> TreeSetExample [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (11-Aug-2023, 4:58:02 pm - 4:58:05 pm) [pid: 16728]'.

Difference between HashSet and TreeSet:

- 1) **HashSet** gives better performance (faster) than **TreeSet** for the operations like add, remove, contains, size etc. HashSet offers constant time cost while TreeSet offers $\log(n)$ time cost for such operations.
- 2) HashSet does not maintain any order of elements while TreeSet elements are sorted in ascending order by default.

Similarities between HashSet and TreeSet:

- 1) Both HashSet and TreeSet do not hold duplicate elements, which means both of these are duplicate free.
- 2) If you want a sorted Set then it is better to add elements to HashSet and then **convert it into TreeSet** rather than creating a TreeSet and adding elements to it.

```
HashSet<Integer> hSet=new HashSet<Integer>();
hSet.add(23);
hSet.add(93);
hSet.add(13);
hSet.add(53);
```

```
TreeSet<Integer> treeSet=new TreeSet<Integer>(hSet);
System.out.println(treeSet);
```

- 3) Both of these classes are non-synchronized, which means they are not thread-safe and should be synchronized explicitly when there is a need for thread-safe operations.

Knowledge check...

Now that we've covered `LinkedHashSet`, `TreeSet`, it's time to test your understanding. The Trainer will conduct a short poll quiz to assess your knowledge on these topics.

1. Which of the following best describes the behavior of a `LinkedHashSet` in Java?

- A) Elements are stored in sorted order based on their natural ordering.
- B) Elements are stored in the order they were inserted.
- C) Elements are stored in a balanced binary search tree.
- D) Elements are stored in a hash table.

2. In a `TreeSet` in Java, elements are:

- A) Stored in insertion order.
- B) Stored in a hash table.
- C) Stored in sorted order based on their natural ordering or a custom comparator.
- D) Stored in a doubly-linked list.

At the end of the quiz, the Trainer will provide feedback and explanations to the participants, enhancing the learning experience and understanding of the content.

Food for thought..

The trainer can ask the students the following question to engage them in a discussion:

Think about a situation where you're designing a simple contact management system. How could the `Map` interface assist in storing names and corresponding phone numbers?.

Java Collections – Map

The Map interface in Java is part of the Java Collections Framework and provides a way to store and manage key-value pairs. Each key is associated with exactly one value, and the keys are unique within a Map. The Map interface is implemented by various classes in Java, each offering different characteristics and behaviors

There are three main implementations of Map interfaces:

HashMap: It makes no guarantees concerning the order of iteration. Implements a hash table for efficient key-based access. Does not guarantee order of entries.

Note:*HashTable: An older synchronized version of HashMap, which is thread-safe but less efficient.*

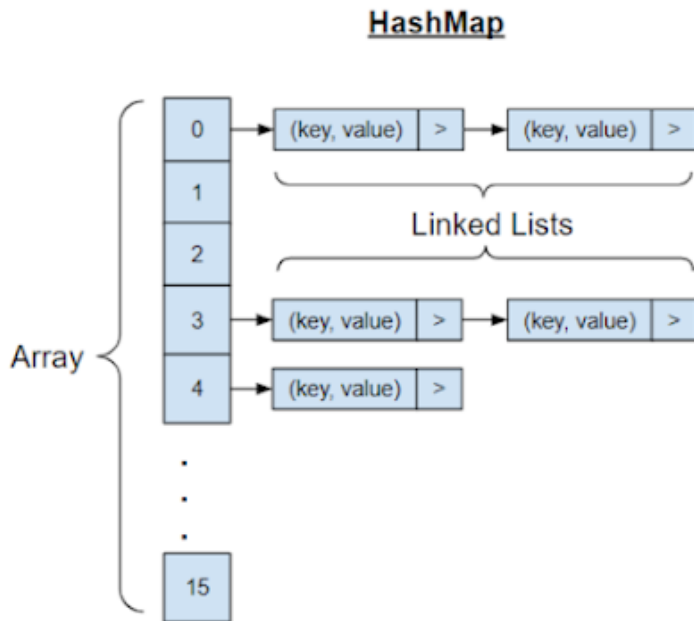
TreeMap: It stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashMap. Implements a self-balancing binary search tree (Red-Black tree) to maintain entries in sorted order.

LinkedHashMap: It orders its elements based on the order in which they were inserted into the set (insertion-order).

HashMap in Java

HashMap is a Map based collection class that is used for storing Key & value pairs, it is denoted as `HashMap<Key, Value>` or `HashMap<K, V>`. This class makes no guarantees as to the order of the map. It is similar to the Hashtable class except that it is unsynchronized and permits nulls(null values and null key).

It is not an ordered collection which means it does not return the keys and values in the same order in which they have been inserted into the HashMap. It does not sort the stored keys and Values. You must need to import `java.util.HashMap` or its super class in order to use the HashMap class and methods.



Map.Entry interface

Map.Entry interface in Java provides certain methods to access the entry in the Map. By gaining access to the entry of the Map we can easily manipulate them. Map.Entry is generic and is defined in the java.util package.

Declaration :

Interface Map.Entry

k -> Key

V -> Value

Methods:

1. equals (Object o) – It compares the object (invoking object) with the Object o for equality.

Syntax :

boolean equals(Object o)

Parameters :

o -> Object to which we want to compare

Returns:

true: if both objects are equals

false: otherwise

2. K getKey() – Returns the key for the corresponding map entry.

Syntax :

K getKey()

Parameters :

Returns:

K -> Returns the corresponding Key of a entry on which it is invoked

Exception –

- **IllegalStateException** is thrown when an entry has been removed from the map.

3. V getValue() – Returns the value for the corresponding map entry.

Syntax :

V getValue()

Parameters :

Returns:

V -> Returns the corresponding value of a entry on which it is invoked

4. int hashCode() – Returns the hashCode for the corresponding map entry.

Syntax :

int hashCode()

Parameters :

Returns:

int -> Returns the hash-code of entry on which it is invoked

5. V setValue(V v) – Sets the value of the map with specified value v

V setValue(V v)

Parameters :

v -> Value which was earlier stored in the entry on which it is invoked

Returns:

int -> Returns the hash-code of a entry on which it is invoked

Exception :

- **ClassCastException** is thrown if the class of the value 'v' is not a correct type for map.

- **NullPointerException** is thrown if 'null' is stored in 'v', and 'null' is not supported by map.
- **UnsupportedOperationException** is thrown if we cannot manipulate the map or the put operation is not supported by the map.
- **IllegalArgumentException** is thrown If there is some problem with the argument i.e v
- **IllegalStateException** is thrown when entry has been removed from the map

Set<Map.Entry> entrySet() – Returns the Set view of the entire map.

Note : This is not a method of Map.entry interface but it is discussed here because this method is useful while working with Map.Entry interface.

Set<Map.Entry> entrySet()

Parameters :

Returns:

Set<Map.Entry> ->: Returns a Set containing the Map.Entry values

```
import java.util.Map;

import java.util.Iterator;

import java.util.Set;

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

        /* This is how to declare HashMap */

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

        /*Adding elements to HashMap*/

        hmap.put(12, "Chaitanya");
```



```
hmap.put(2, "Rahul");

hmap.put(7, "Singh");

hmap.put(49, "Ajeet");

hmap.put(3, "Anuj");


/* Display content using Iterator*/

Set set = hmap.entrySet();

Iterator iterator = set.iterator();

while(iterator.hasNext()) {

    Map.Entry mentry = (Map.Entry)iterator.next();

    System.out.print("key is: "+ mentry.getKey() + " & Value is: ");

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

}


/* Get values based on key*/

String var= hmap.get(2);

System.out.println("Value at index 2 is: "+var);


/* Remove values based on key*/

hmap.remove(3);

System.out.println("Map key and values after removal:");

Set set2 = hmap.entrySet();

Iterator iterator2 = set2.iterator();
```

```

while(iterator2.hasNext()) {

    Map.Entry mentry2 = (Map.Entry)iterator2.next();

    System.out.print("Key is: "+mentry2.getKey() + " & Value is: ");

    System.out.println(mentry2.getValue());

}

}

```

Output:

```

MapEx.java - Eclipse IDE
Window Help
History Synchronize Git Staging Git Reflog Properties JUnit Console SonarLint Report
<terminated> HashMap [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (14-Aug-2023, 2:19:50 pm - 2:19:51 pm) [pid: 16348]
key is: 49 & Value is: Ajeet
key is: 2 & Value is: Rahul
key is: 3 & Value is: Anuj
key is: 7 & Value is: Singh
key is: 12 & Value is: Chaitanya
Value at index 2 is: Rahul
Map key and values after removal:
Key is: 49 & Value is: Ajeet
Key is: 2 & Value is: Rahul
Key is: 7 & Value is: Singh
Key is: 12 & Value is: Chaitanya

```

Internal working of HashMap:

The internal working of a HashMap in Java involves concepts like hashing, buckets, hash collisions, and resizing. Understanding these concepts is crucial for efficient use of HashMap.

Here's a high-level overview of how a HashMap works internally:

Hashing:

When you insert a key-value pair into a HashMap, the HashMap calculates a hash code for the key using its `hashCode()` method. This hash code is used to determine the index (bucket) where the key-value pair will be stored.

Bucket Array:

A HashMap maintains an array of buckets to store key-value pairs. The size of this array is initially set and can grow dynamically as the map fills up. Each bucket can hold multiple key-value pairs, forming a linked list or a tree (for Java 8+ in case of hash collisions).

Index Calculation:

To find the index of the bucket for a key, the hash code is processed using an algorithm that ensures the index falls within the array bounds. Typically, a bitwise operation is performed on the hash code to reduce it to a valid index within the array.

Hash Collisions:

Hash collisions occur when two different keys have the same hash code and need to be stored in the same bucket. To handle collisions, modern HashMap implementations use linked lists (before Java 8) or trees (from Java 8 onward) within buckets to store multiple key-value pairs with the same hash code.

Resizing:

As the number of key-value pairs increases, the load factor (ratio of filled buckets to total buckets) of the HashMap increases. When the load factor exceeds a certain threshold, the HashMap automatically resizes its bucket array. This involves creating a new, larger array and rehashing all existing key-value pairs into the new buckets. This process helps maintain efficient performance by reducing the likelihood of hash collisions.

Balancing Mechanisms (Java 8+):

To improve performance in the presence of hash collisions, HashMap in Java 8+ switches from using linked lists to using trees in buckets when the number of elements in a bucket exceeds a certain threshold. This enhances performance for hash maps with skewed data distribution.

Key Equality and Retrieval:

When you try to retrieve a value using a key, the HashMap calculates the hash code of the key and then searches the appropriate bucket to find the matching key. If a match is found, the associated value is returned.

Iterating Through Entries:

When iterating through the entries of a HashMap, you traverse the array of buckets and then iterate through the linked list or tree within each bucket to access the key-value pairs.

Performance Considerations:

A well-distributed hash code function and an appropriate initial capacity help maintain a balanced distribution of key-value pairs. Keeping the load factor reasonable also prevents excessive collisions and resizing operations.

LinkedHashMap

LinkedHashMap is a Hash table and linked list implementation of the Map interface, with predictable iteration order. This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order).

This class is different from HashMap and TreeMap:

- **HashMap** doesn't maintain any order.
- **TreeMap** sorts the entries in ascending order of keys.
- **LinkedHashMap** maintains the insertion order.

Important Features of a LinkedHashMap are listed as follows:

- A LinkedHashMap contains values based on the key. It implements the Map interface and extends the HashMap class.
- It contains only unique elements.
- It may have one null key and multiple null values.
- It is non-synchronized.
- It is the same as HashMap with an additional feature that it maintains insertion order. For example, when we run the code with a HashMap, we get a different order of elements.

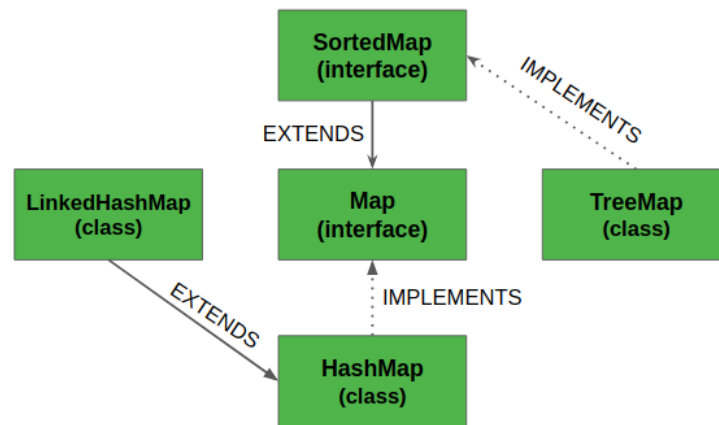
Declaration:

```
public class LinkedHashMap<K,V> extends HashMap<K,V> implements Map<K,V>
```

Here, **K** is the key Object type and **V** is the value Object type

- **K** – The type of the keys in the map.
- **V** – The type of values mapped in the map.

It implements Map<K, V> interface, and extends HashMap<K, V> class. Though the Hierarchy of LinkedHashMap is as depicted in below media as follows:



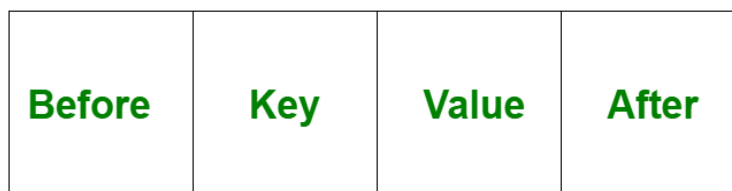
MAP Hierarchy in Java

How LinkedHashMap Works Internally?

A **LinkedHashMap** is an extension of the **HashMap** class and it implements the **Map** interface. Therefore, the class is declared as:

```
public class LinkedHashMap  
extends HashMap  
implements Map
```

In this class, the data is stored in the form of nodes. The implementation of the **LinkedHashMap** is very similar to a doubly-linked list. Therefore, each node of the **LinkedHashMap** is represented as:



- **Hash:** All the input keys are converted into a hash which is a shorter form of the key so that the search and insertion are faster.
- **Key:** Since this class extends **HashMap**, the data is stored in the form of a key-value pair. Therefore, this parameter is the key to the data.
- **Value:** For every key, there is a value associated with it. This parameter stores the value of the keys. Due to generics, this value can be of any form.

- **Next:** Since the LinkedHashMap stores the insertion order, this contains the address to the next node of the LinkedHashMap.
- **Previous:** This parameter contains the address to the previous node of the LinkedHashMap.

Synchronized LinkedHashMap

The implementation of LinkedHashMap is not synchronized. If multiple threads access a linked hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the map. If no such object exists, the map should be “wrapped” using the **Collections.synchronizedMap** method. This is best done at creation time, to prevent accidental unsynchronized access to the map:

```
Map m = Collections.synchronizedMap(new LinkedHashMap(...));
```

Constructors of LinkedHashMap Class

In order to create a LinkedHashMap, we need to create an object of the LinkedHashMap class. The LinkedHashMap class consists of various constructors that allow the possible creation of the ArrayList. The following are the constructors available in this class:

1. LinkedHashMap(): This is used to construct a default LinkedHashMap constructor.

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<K, V>();
```

2. LinkedHashMap(int capacity): It is used to initialize a particular LinkedHashMap with a specified capacity.

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<K, V>(int capacity);
```

3. LinkedHashMap(Map<? extends K,? extends V> map): It is used to initialize a particular LinkedHashMap with the elements of the specified map.

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<K, V>(Map<? extends K,? extends V> map);
```

4. LinkedHashMap(int capacity, float fillRatio): It is used to initialize both the capacity and fill ratio for a LinkedHashMap. A fillRatio also called as **loadFactor** is a metric that determines when to increase the size of the LinkedHashMap automatically. By default,

this value is 0.75 which means that the size of the map is increased when the map is 75% full.

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<K, V>(int capacity, float fillRatio);
```

5. LinkedHashMap(int capacity, float fillRatio, boolean Order): This constructor is also used to initialize both the capacity and fill ratio for a LinkedHashMap along with whether to follow the insertion order or not.

```
LinkedHashMap<K, V> lhm = new LinkedHashMap<K, V>(int capacity, float fillRatio, boolean Order);
```

Here, For the Order attribute, true is passed for the last access order and false is passed for the insertion order.

Methods of LinkedHashMap

METHOD	DESCRIPTION
containsValue(Object value)	Returns true if this map maps one or more keys to the specified value.
entrySet()	Returns a Set view of the mappings contained in this map.
get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
keySet()	Returns a Set view of the keys contained in this map.
removeEldestEntry(Map.Entry<K,V> eldest)	Returns true if this map should remove its eldest entry.
values()	Returns a Collection view of the values contained in this map.

```

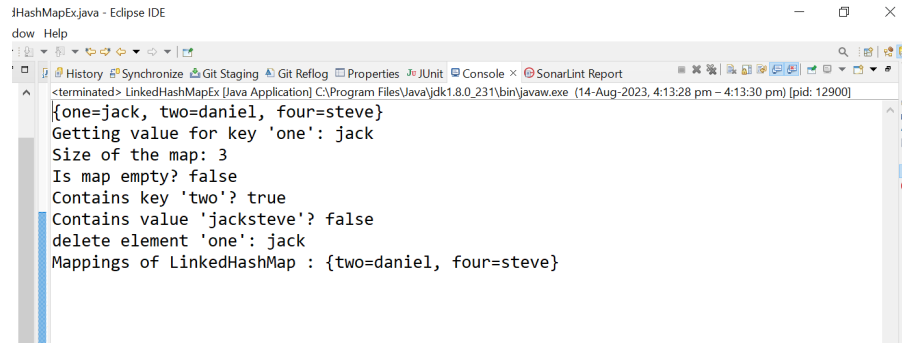
package com.anudip;
import java.util.LinkedHashMap;
public class LinkedHashMapEx {
    public static void main(String[] args) {
        // Creating an empty LinkedHashMap
        LinkedHashMap<String, String> lhm
        = new LinkedHashMap<String, String>();
        // Adding entries in Map
        // using put() method
        lhm.put("one", "jack");
        lhm.put("two", "daniel");
        lhm.put("four", "steve");
        // Printing all entries inside Map
        System.out.println(lhm);
        // Note: It prints the elements in same order
        // as they were inserted
        // Getting and printing value for a specific key
        System.out.println("Getting value for key 'one': "
        + lhm.get("one"));
        // Getting size of Map using size() method
        System.out.println("Size of the map: "
        + lhm.size());
        // Checking whether Map is empty or not
        System.out.println("Is map empty? "
        + lhm.isEmpty());
        // Using containsKey() method to check for a key
        System.out.println("Contains key 'two'? "
        + lhm.containsKey("two"));
        // Using containsKey() method to check for a value
        System.out.println(
        "Contains value 'jack"
        + "steve'? "
        + lhm.containsValue("jack"
        + ".steve"));
        // Removing entry using remove() method
        System.out.println("delete element 'one': "
        + lhm.remove("one"));
        // Printing mappings to the console
        System.out.println("Mappings of LinkedHashMap : "
        + lhm);
    }
}

```



```
}  
}
```

Output



```
JHashMapEx.java - Eclipse IDE  
dow Help  
History Synchronize Git Staging Git Reflog Properties JUnit Console SonarLint Report  
<terminated> LinkedHashMapEx [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (14-Aug-2023, 4:13:28 pm - 4:13:30 pm) [pid: 12900]  
{one=jack, two=daniel, four=steve}  
Getting value for key 'one': jack  
Size of the map: 3  
Is map empty? false  
Contains key 'two'? true  
Contains value 'jacksteve'? false  
delete element 'one': jack  
Mappings of LinkedHashMap : {two=daniel, four=steve}
```

Operation 1: Changing/Updating Elements

After adding elements, if we wish to change the element, it can be done by again adding the element using the put() method. Since the elements in the LinkedHashMap are indexed using the keys, the value of the key can be changed by simply re-inserting the updated value for the key for which we wish to change.

// Java Program to Demonstrate Updation of Elements
// of LinkedHashMap

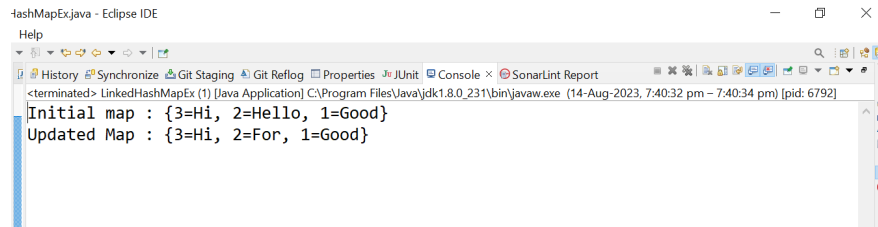
```
package queueExamples;  
import java.util.*;  
public class LinkedHashMapEx {  
    public static void main(String[] args) {  
        // Initialization of a LinkedHashMap  
        // using Generics  
        LinkedHashMap<Integer, String> hm  
        = new LinkedHashMap<Integer, String>();  
        // Inserting mappings into Map  
        // using put() method  
        hm.put(3, "Hi");  
        hm.put(2, "Hello");  
        hm.put(1, "Good");  
        // Printing mappings to the console  
        System.out.println("Initial map : " + hm);  
        // Updating the value with key 2
```

```

hm.put(2, "For");
// Printing the updated Map
System.out.println("Updated Map : " + hm);
    }
}

```

Output



Operation 2: Removing Element

In order to remove an element from the LinkedHashMap, we can use the `remove()` method. This method takes the value of the key as input, searches for the existence of such key and then removes the mapping for the key from this LinkedHashMap if it is present in the map. Apart from that, we can also remove the first entered element from the map if the maximum size is defined.

Example

// Java program to Demonstrate Removal of Elements
 // from LinkedHashMap

```

// Importing utility classes
import java.util.*;

// Main class
// RemovingMappingsFromLinkedHashMap
package queueExamples;
import java.util.*;

public class LinkedHashMapEx {
    public static void main(String[] args) {
        // Initialization of a LinkedHashMap
        // using Generics
        LinkedHashMap<Integer, String> names
        = new LinkedHashMap<Integer, String>();
        // Inserting the Elements
        // using put() method

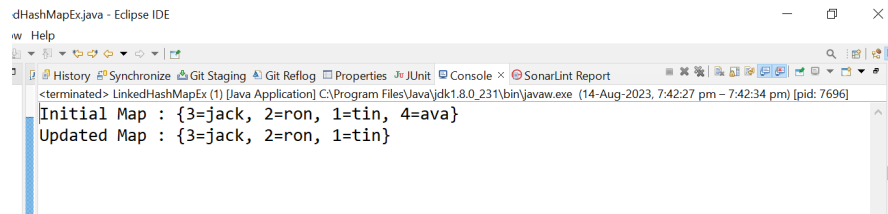
```

```

names.put(3, "jack");
names.put(2, "ron");
names.put(1, "tin");
names.put(4, "ava");
// Printing the mappings to the console
System.out.println("Initial Map : " + names);
// Removing the mapping with Key 4
names.remove(4);
// Printing the updated map
System.out.println("Updated Map : " + names);
    }
}

```

Output:



Operation 3: Iterating through the LinkedHashMap

There are multiple ways to iterate through the LinkedHashMap. The most famous way is to use a for-each loop over the set view of the map (fetched using map.entrySet() instance method). Then for each entry (set element) the values of key and value can be fetched using the getKey() and the *getValue()* method.

```

// Java program to demonstrate
// Iterating over LinkedHashMap

```

```

// Importing required classes
import java.util.*;

```

```

// Main class
// IteratingOverLinkedHashMap
package queueExamples;
import java.util.*;
public class LinkedHashMapEx {
    public static void main(String[] args) {

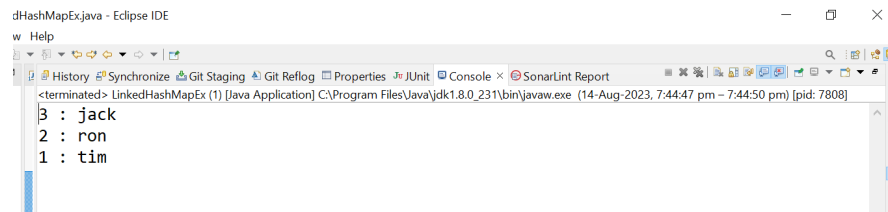
```

```

// Initialization of a LinkedHashMap
// using Generics
LinkedHashMap<Integer, String> hm
= new LinkedHashMap<Integer, String>();
// Inserting elements into Map
// using put() method
hm.put(3, "jack");
hm.put(2, "ron");
hm.put(1, "tim");
// For-each loop for traversal over Map
for (Map.Entry<Integer, String> mapElement :
hm.entrySet()) {
Integer key = mapElement.getKey();
// Finding the value
// using getValue() method
String value = mapElement.getValue();
// Printing the key-value pairs
System.out.println(key + " : " + value);
}
}
}

```

Output:



The screenshot shows the Eclipse IDE interface. The console window at the bottom displays the output of the program, which is a list of key-value pairs from the LinkedHashMap. The output is as follows:

```

3 : jack
2 : ron
1 : tim

```

The console window also shows the command prompt text: "terminated> LinkedHashMapEx (1) [Java Application] C:\Program Files\Java\jdk1.8.0_231\bin\javaw.exe (14-Aug-2023, 7:44:47 pm - 7:44:50 pm) [pid: 7808]".