Java Collections

By

Praveen Oruganti
Linktree: https://linktr.ee/praveenoruganti
Email: praveenorugantitech@gmail.com

Array

An array is an index collection of fixed no of homogeneous data elements. The main advantage of array is we can represent multiple value by using single variable, so that readability of code will be improved.

Limitations of Array

1. Array are fixed size i.e.. we create an array there is no chance of increasing or decreasing the size based on our requirement due to this, to use array concept compulsory we should know the size in advance, which may not possible always.

2. Array concept is not implement based on standard data structure, hence readymade method support is not available. For every requirements we have to write the code explicitly.

3. We can hold only homogeneous data type elements. We can solve this problem by using object type array.

Collection

If we want to represent a group of individual object as a single entities then we should go for collection.

Difference between array and collection:

Array

- ✓ Array are fixed in size.
- ✓ Not have proper memory management.
- ✓ Performance is relatively good.
- ✓ No underlying data structure.
- ✓ Readymade method support not available.
- ✓ Primitive and object bot type available.
- ✓ Only Homogeneous object.

Collection

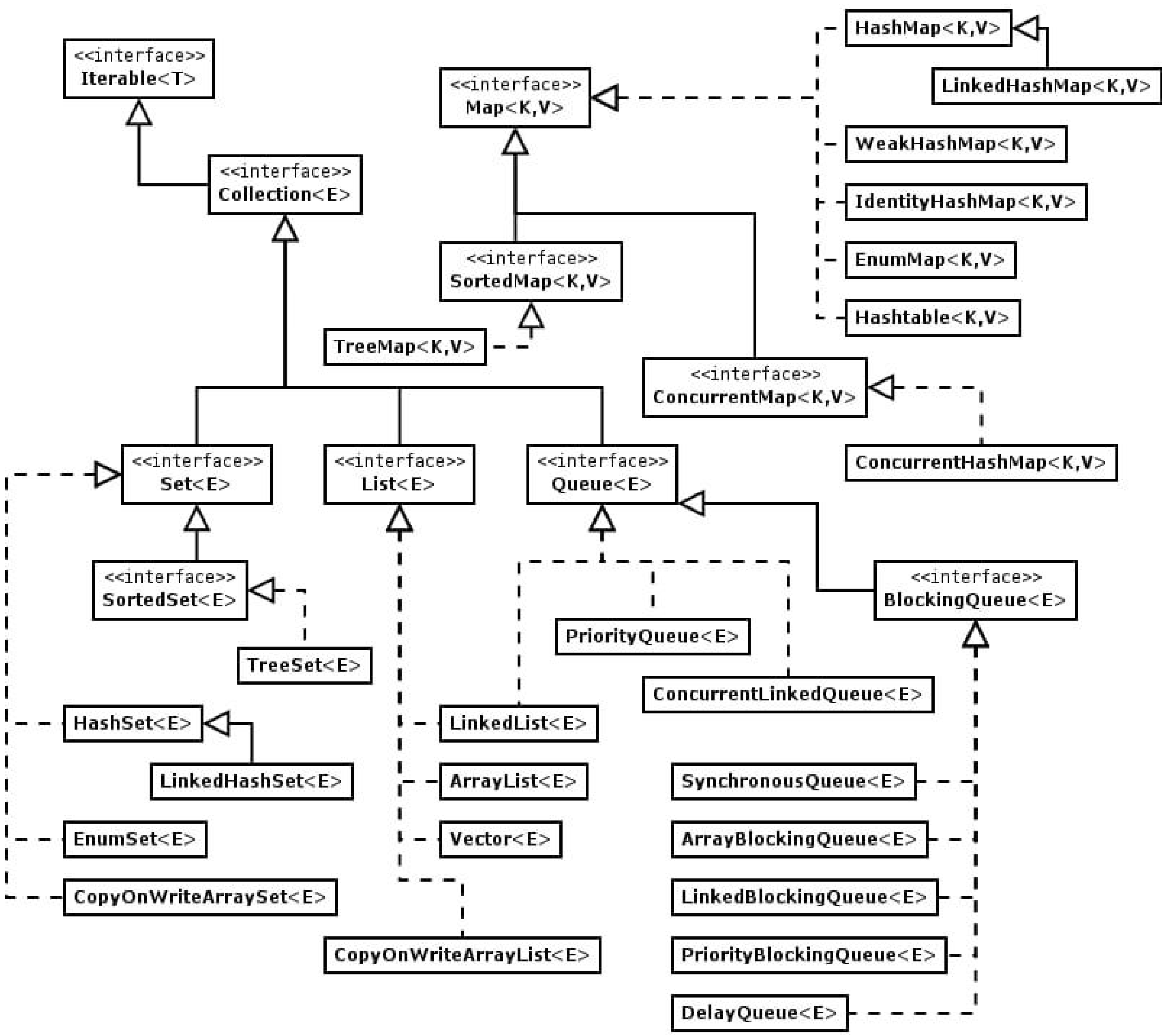
- ✓ Collection are growable in size.
- ✓ Good memory management.
- ✓ Collection are not good in performance.
- ✓ Both Homogeneous and Heterogeneous object allowed.
- ✓ Readymade method available.
- ✓ Only objects are allowed.
- ✓ Standard data structure.

Difference between Collection and Collections

Collection is an interface, if we want to represent a group of individual object as a single entity then we should go for Collection

Collections is an utility class present in java.util package. It defines several utility method for collection object like sorting searching etc.
Collections is an utility class and some of the important methods are
a) sort b) EMPTY_LIST c) binarySearch d) reverse e) shuffle f) swap g) copy h) min i) max j) replaceAll h) unmodifiableCollection i) forEach j) removeIf k) stream l) parallelstream m) synchronizedCollection

Collection Hierarchy



Interfaces in Collections

1.Collection: the foundational interface of the Collections Framework.

2.List: extends Collection. Defines a collection that stores an ordered sequence of elements.

3.Queue: extends Collection. Defines a collection where insertion and removal each occur only at a single end. Typically elements are ordered as a FIFO queue, although this can be altered to accommodate priority queues or LIFO stacks.

4.Deque: extends Queue. Short for “double ended queue”, it defines a queue where insertion and removal can occur at either end.

5.Set: extends Collection. Defines a collection that does not allow duplicate elements.

6.SortedSet: extends Set. Defines a set in which elements are sorted either according to their natural ascending order or by the use of a Comparator.

7.NavigableSet: extends SortedSet. Defines a sorted set which supports searching for elements based on closest matches. For instance, this interface defines the higher(E e) method, which returns the smallest element larger than the argument e.

Non-Abstract Classes in Collections

1.ArrayList: extends AbstractList, and implements the List interface. Similar to the Vector class described above, it offers a more fully featured, resizable alternative to Java’s native arrays.

2.LinkedList: extends AbstractSequentialList, and implements the Deque interface. Creates a doubly Linked list. Linked lists are somewhat akin to arrays, but don’t require continuous blocks of memory. This is the class you’ll most likely want to use for stacks.

3.ArrayDeque: extends AbstractCollection, and implements the Deque interface. This is the class you’ll most likely want to use for queues, double-ended or otherwise, though it can also be used for stacks.

4.PriorityQueue: extends AbstractQueue, and implements the Queue interface. As the name implies, you’ll want to use it for queues that are ordered according to priority.

5.HashSet: extends AbstractSet, and implements the Set interface. It is a set whose elements are backed by a HashMap under the hood. Due to the unpredictable nature of hashing, its elements are not reliably ordered — thus, it is useful primarily for sets which do not need to be sorted.

6.LinkedHashSet: extends HashSet, and implements the Set interface. It is similar to HashSet, but also maintains a linked list of the entries in the order in which they were inserted, allowing you to iterate through them thusly.

7.TreeSet: extends AbstractSet, and implements the NavigableSet interface. It is similar to HashSet, but is backed by a TreeMap. This allows allows the set to be sorted, either in natural ascending ordering or by use of a Comparator.

8.EnumSet: extends AbstractSet. It is a set which is specially designed to work with enum types, and where all members of the set must come from the same enum.

Linktree: https://linktr.ee/praveenoruganti
Email: praveenorugantitech@gmail.com

Abstract Classes in Collections

1.AbstractCollection: Implements most of the Collection interface.

2.AbstractList: Extends AbstractCollection and implements most of the List interface.

3.AbstractSequentialList: Extends AbstractList. As the name implies, it is meant to facilitate sequential, rather than random, access of its elements.

4.AbstractQueue: Extends AbstractCollection and implements some of the Queue interface.

5.AbstractSet: Extends AbstractCollection and implements most of the Set interface.

List

List interface accepts null, duplicates and maintain insertion order.

ArrayList

ArrayList is implemented as a resizable array. As more elements are added to ArrayList, its size is increased dynamically. It's elements can be accessed directly by using the get and set methods, since ArrayList is essentially an array.ArrayList is a better choice if your program is thread-safe.ArrayList grow 50% of its size each time.
ArrayList implements RandomAccess Marker interface to indicate that they support fast (generally constant time) **random access**.

LinkedList

LinkedList is implemented as a double linked list. Its performance on add and remove is better than Arraylist, but worse on get and set methods.LinkedList, however, also implements Queue interface which adds more methods than ArrayList and Vector, such as offer(), peek(), poll(), etc.

Vector

Vector is similar with ArrayList, but it is synchronized.Vector each time doubles its array size.

Set

Set interface will not accept duplicates.

Hashset

Hashset allows one null value and doesnt maintain insertion order. It internally uses HashMap as a backing datastructure with key as generic type E which is our element and value as Object class type which is denoted by static field **PRESENT**.

LinkedHashSet

LinkedHashSet is an extended version of Hashset and it internally uses LinkedHashMap.It maintains insertion order.

TreeSet

TreeSet internally uses TreeMap and it maintains natural order and it doesn't allow null value. It implements NavigableSet interface as well as SortedSet interface.
When you are inserting objects in TreeSet you need to override equals and hashcode.
Lets see Employee class which is overriding equals and hashcode.

```
class Employee implements Comparable<Employee> {


    private int empId;

    private String empName;

    private int empAge;

    private double empSal;

    public Employee(int empId, String empName, int empAge, double empS
al) {

        this.empId = empId;

        this.empName = empName;

        this.empAge = empAge;

        this.empSal = empSal;

    }

    public int getEmpId() {

        return empId;

    }

    public String getEmpName() {
```


Linktree: https://linktr.ee/praveenoruganti
Email: praveenorugantitech@gmail.com

```java
        return empName;
    }
    public int getEmpAge() {
        return empAge;
    }
    public double getEmpSal() {
        return empSal;
    }
```

```java
    @Override
    public int compareTo(Employee emp) {
        if (this.empId == emp.empId)
            return 0;
        else if (this.empId > emp.empId)
            return +1;
        else
            return -1;
    }
    @Override
    public String toString() {
        return "Employee [empId=" + empId + ", empName=" + empName + ",
 empAge=" + empAge + ", empSal=" + empSal + "]";
    }
}
```


Linktree: https://linktr.ee/praveenoruganti
Email: praveenorugantitech@gmail.com

Comparing Sets

	Data Structure	Sorting	Iterator	Nulls?
HashSet	Hash table	No	Fail-fast	Yes
Linked HashSet	Hash table + linked list	Insertion Order	Fail-fast	Yes
EnumSet	Bit vector	Natural Order	Weakly consistent	No
TreeSet	Red-black tree	Sorted	Fail-fast	Depends
CopyOnWrite ArraySet	Array	No	Snapshot	Yes
Concurrent SkipListSet	Skip list	Sorted	Weakly consistent	No

Difference between List and Set

	List	Set
Duplicates	YES	NO
Order	ORDERED	DEPENDS ON IMPLEMENTATION
Positional Access	YES	NO

Performance of List and Set

Name	Insertion	Deletion	Contains()	get()	peek()
ArrayList	At ends - O(1)	At any index - O(n)	O(n)	O(1)	-
LinkedList	At ends - O(1)	At any index - O(n)	O(n)	O(n)	-
HashSet	O(1)	O(1)	O(1)	-	-
LinkedHashSet	O(1)	O(1)	O(1)	-	-
TreeSet	O(logN)	O(logN)	O(logN)	-	-
EnumSet	O(1)	O(1)	O(1)	-	-
PriorityQueue	O(logN)	O(logN)	O(n)	-	O(1)
ArrayDeque	At ends - O(1)	At ends - O(1)	O(n)	-	O(1)

Difference between Array and ArrayList

	Size	They can hold	Iteration	How to get size?	Generics	Type Safe	Multi-dimensional	How to add elements?
Array	Fixed	Primitives as well as objects	Only through *for* loop or *for-each* loop	*Length* attribute	Doesn't support	No	Yes	Using assignment operator
ArrayList	Re-sizable	Only objects	Iterators or *for* loop or *for-each* loop	*size()* method	supports	Yes	No	Using *add()* method

Difference between ArrayList and Vector

ArrayList	Vector
ArrayList is **not synchronized**.	Vector is **synchronized**.
ArrayList is **not a legacy** class, it is introduced in JDK 1.2.	Vector is a **legacy** class.
ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.
ArrayList uses **Iterator** interface to traverse the elements.	Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also.
ArrayList **increments 50%** of current array size if number of element exceeds from its capacity.	Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity.

Difference between ArrayList and LinkedList

Structure	**ArrayList** ArrayList is an index based data structure where each element is associated with an index.	**LinkedList** Elements in the LinkedList are called as nodes, where each node consists of three things – Reference to previous element, Actual value of the element and Reference to next element.
Insertion And Removal	Insertions and Removals in the middle of the ArrayList are very slow. Because after each insertion and removal, elements need to be shifted.	Insertions and Removals from any position in the LinkedList are faster than the ArrayList. Because there is no need to shift the elements after every insertion and removal. Only references of previous and next elements are to be changed.
Retrieval (Searching or getting an element)	Insertion and removal operations in ArrayList are of order O(n). Retrieval of elements in the ArrayList is faster than the LinkedList . Because all elements in ArrayList are index based.	Insertion and removal in LinkedList are of order O(1). Retrieval of elements in LinkedList is very slow compared to ArrayList. Because to retrieve an element, you have to traverse from beginning or end (Whichever is closer to that element) to reach that element.
Random Access	Retrieval operation in ArrayList is of order of O(1). ArrayList is of type Random Access. i.e elements can be accessed randomly.	Retrieval operation in LinkedList is of order of O(n). LinkedList is not of type Random Access. i.e elements can not be accessed randomly. you have to traverse from beginning or end to reach a particular element.
Usage	ArrayList can not be used as a Stack or Queue.	LinkedList, once defined, can be used as ArrayList, Stack, Queue, Singly Linked List and Doubly Linked List.
Memory Occupation	ArrayList requires less memory compared to LinkedList. Because ArrayList holds only actual data and it's index.	LinkedList requires more memory compared to ArrayList. Because, each node in LinkedList holds data and reference to next and previous elements.
When To Use	If your application does more retrieval than the insertions and deletions, then use ArrayList.	If your application does more insertions and deletions than the retrieval, then use LinkedList.

Linktree: https://linktr.ee/praveenoruganti
Email: praveenorugantitech@gmail.com

Iterator interface

An iterator is an interface that iterates the elements. It is used to traverse the list and modify the elements.

Iterator interface has three methods which are mentioned below:

1.public boolean hasNext() — This method returns true if the iterator has more elements.
2.public object next() — It returns the element and moves the cursor pointer to the next element.
3.public void remove() — This method removes the last elements returned by the iterator.

Difference between Enumerator and Iterator

	Introduced From	Methods	Main Difference	Fail-Fast Vs Fail-Safe	Safe And Secure
Enumeration	JDK 1.0	*hasMoreElements()* *nextElement()*	You can only traverse the collection.	Fail-Safe	No
Iterator	JDK 1.2	*hasNext()* *next()* *remove()*	You can also remove an element while iterating the collection.	Fail-Fast	Yes

Difference between Iterator and ListIterator

Iterator	ListIterator
Can do remove operation only on elements	Can remove, add and replace elements
Method is remove()	Methods are remove(), add() and set()
iterator() method returns an object of Itertor	listIterator() method returns an object of ListItertor
iterator() method is available for all collections. That is, Iterator can be used for all collection classes	listIterator() method is available for those collections that implement List interface. That is, descendents of List interface only can use ListIterator

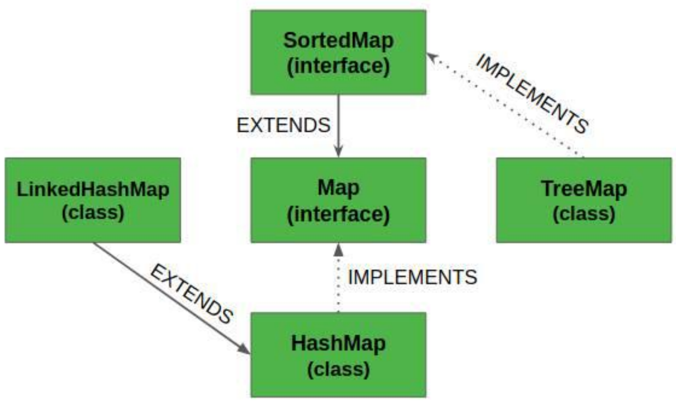
Fail fast vs Fail safe iterators

Iterators in java give us the facility to traverse over the Collection objects. Iterators returned by the Collection are either fail-fast in nature or fail-safe in nature.

Fail-Fast iterators immediately throw ConcurrentModificationException if a collection is modified while iterating over it. For example,
Iterators returned by ArrayList, Vector, HashMap.

Where as Fail-Safe iterators don't throw any exceptions if a collection is modified while iterating over it. Because, they operate on the clone of the collection, not on the actual collection. For example, Iterator returned by ConcurrentHashMap.

Map interface



MAP Hierarchy in Java

Interfaces in Maps

1.Map: defines an object that maps keys to values.

2.SortedMap: extends Map. Defines a map that supports the ordering of its keys, either in natural ascending order or by the use of a Comparator.

3.NavigableMap: extends SortedMap. Defines a sorted map which supports searching for entries based on closest matches (to the keys, not the values).

Non-Abstract Classes in Maps

1.HashMap: extends AbstractMap, and implements the Map interface. It is roughly equivalent to the HashTable legacy class we covered earlier, but newer, better, faster, stronger.

2.LinkedHashMap: extends HashMap, and implements the Map interface. Like the LinkedHashSet, we covered earlier, this one also maintains a linked list under the hood that allows you iterate over entries in the order in which they were inserted.

3.TreeMap: extends AbstractMap, and implements the NavigableMap interface. It organizes keys via a Red-Black tree, allowing them to be sorted either in natural ascending order or by the use of a Comparator.

4.EnumMap: extends AbstractMap. A specialized map meant specifically for working with enum types, where all the keys in the map must come from the same enum.

5.WeakHashMap: extends AbstractMap, and implements the Map interface. Essentially identical to HashMap, but with weak keys. This means that entries are susceptible to being discarded by Java's garbage collector when their keys are no longer in ordinary use.

6.IdentityHashMap: extends AbstractMap, and implements the Map interface. Again, essentially identical to HashMap, but uses reference-equality instead of object-equality when comparing keys or values. The API explicitly says this class is not for general usage.

HashMap

HashMap class implements the map interface by using a hash table.

1. HashMap don't contain duplicate keys and can contain duplicate values.
2. HashMap class may have one null key and multiple null values.
3. HashMap class is not synchronized.
4. HashMap class doesn't maintain order
5. The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

```
HashMap<Integer, String> hm = new HashMap<Integer, String>();
        hm.put(100, "Praveen");
        hm.put(101, "Prasad");
        hm.put(102, "Kiran");

        hm.putIfAbsent(103, "Krishna");
        hm.put(104, "Ravi");
        hm.remove(100);
        hm.replace(101, "Praveen");
        for (Map.Entry<Integer, String> m : hm.entrySet()) {
            System.out.println(m.getKey() + " " + m.getValue());
```

```
        }
```

Output
101 Praveen
102 Kiran
103 Krishna
104 Ravi

HashMap internal working

What is Hashing
It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

HashMap uses a technique called **Hashing**.
It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value.
equals(): It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
hashCode(): This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
Buckets: Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

For example,we want to insert three (Key, Value) pair in the HashMap.

```
HashMap<String, Integer> map = new HashMap<>();
        map.put("Praveen", 34);
        map.put("Prasad", 35);
        map.put("Kiran", 38);
```

To store the key in memory we have to calculate the index.

For example, Praveen index is 4

If Prasad index is 4, then there is a hash collision hence it will be next node for Praveen in index 4 itself.

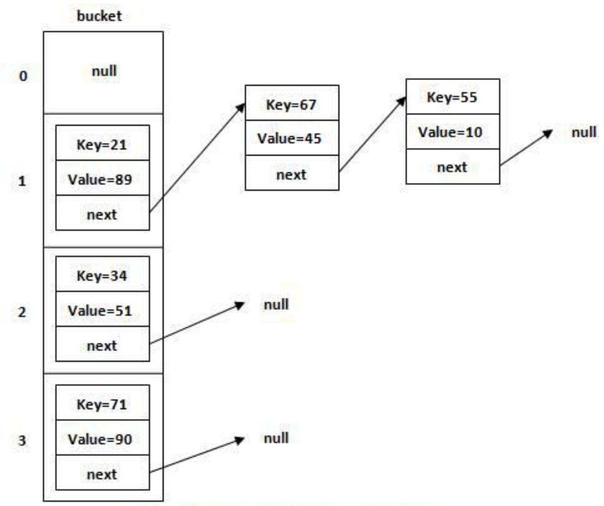


Figure: Allocation of nodes in Bucket

To use HashMap in multi threaded environment, you must write your code in synchronized block or use any external Lock implementation but in that case there is high chances of error and deadlock situations, if proper care is not taken care of.
In short it is not advisable to use HashMap in multi threaded environment. Instead use any of the similar thread safe collections like HashTable, Collections.SynchronizedMap or ConcurrentHashMap.

Though all of them are thread safe but ConcurrentHashMap provides better performance.

HashTable

HashTable is a legacy class uses synchroized methods to achive thread safety but at a time only one thread can read or write in other words thread acquires lock on entire HashTable instance hence performance is slow. HashTable doesnt allow null keys or values where as HashMap allows one null key and multiple null values.

Collections.SynchronizedMap

SynchronizedMap is static inner class of utility class Collections.It is quite similar to HashTable and it allows acquires lock on entire Map instance. It may allow null keys and null values based on the original collection class being passed to it.

ConcurrentHashMap

HashTable and Collections.SynchronizedMap both acquires lock on entire Map instance which provides thread safety but not good performance as at a time only one thread object can access that Map instance.
To overcome this issue ConcurrentHashMap was introduced in Java 5.
More than one thread can read and write concurrently in ConcurrentHashMap and it still provides thread safety.
ConcurrentHashMap divides the Map instance into different segments and each thread acquires lock on each segment.
By default it allows 16 threads to access simultaneously without external synchronization i.e.. default concurrency level is 16. We can also increase or decrease the concurrency level by using below constructor
ConcurrentHashMap(int intialCapacity,fload loadfactor,int concurrencyLevel).
Please note multiple threads cant write on same segment but multiple threads can read from the same segment.
ConcurrentHashMap doesnt allow null keys and null values.

LinkedHashMap

LinkedHashMap is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface. LinkedHashMap maintains insertion order apart from the above HashMap features.

TreeMap

TreeMap is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.It implements the NavigableMap interface and extends AbstractMap class.TreeMap cannot have a null key but can have multiple null values.TreeMap maintains ascending order by default.

Hashtable

Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

- A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
- Java Hashtable class contains unique elements.
- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

IdentityHashMap

- It is exactly same as HashMap except the following difference.
- In case of HashMap, JVM will use .equals(-) method to identify duplicate keys, which is meant for content comparison.
- But in case of IdentityHashMap, JVM will use == operator to identify duplicate keys which is meant for reference comparison.

WeakHashMap

- It is exactly same as HashMap except the following difference.
- In case of HashMap, if an object associated with HashMap then it is not eligible for garbage collection,eventhough it doesn't contain any external references. i.e., HashMap dominates Garbage Collector.
- But in case of WeakHashMap, if an object doesn't contain any references then it is always eligible for Garbage Collector eventhough it is associated with WeakHashMap. ie., Garbage Collector dominates WeakHashMap.

How to create a Custom Map(key/value pair)

```java
import java.util.ArrayList;

import java.util.List;


public class MyMap {

    class Container {

        Object key;

        Object value;

        public void insert(Object k, Object v) {

            this.key = k;

            this.value = v;

        }

    }

    private Container c;

    private List<Container> recordList;

    public MyMap() {

        this.recordList = new ArrayList<Container>();

    }
```

```java
public void put(Object k, Object v) {
    this.c = new Container();
    c.insert(k, v);
    // check for the same key before adding
    for (int i = 0; i < recordList.size(); i++) {
        Container c1 = recordList.get(i);
        if (c1.key.equals(k)) {
            // remove the existing object
            recordList.remove(i);
            break;
        }
    }
    recordList.add(c);
}
public Object get(Object k) {
    for (int i = 0; i < this.recordList.size(); i++) {
        Container con = recordList.get(i);
        if (k.toString() == con.key.toString()) {
            return con.value;
        }
    }
    return null;
}
public static void main(String[] args) {
    MyMap hm = new MyMap();
    hm.put("1", "1");
    hm.put("2", "2");
    hm.put("3", "3");
```

```java
        System.out.println(hm.get("3"));
        hm.put("3", "4");
        System.out.println(hm.get("1"));
        System.out.println(hm.get("3"));
        System.out.println(hm.get("8"));
    }
}
```

Difference between Map Implementations

Property	HashMap	TreeMap	LinkedHashMap	HashTable
Iteration Order	Random	Sorted according to natural **order of keys**	Sorted according to the **insertion order**.	Random
Efficiency: Get, Put, Remove, ContainsKey	$O(1)$	$O(\log(n))$	$O(1)$	$O(1)$
Null keys/values	allowed	Not-allowed*	allowed	Not-allowed
Interfaces	Map	Map, SortedMap, NavigableMap	Map	Map
Synchronized	Not instead use Collection.synchronizedMap(new HashMap())			Yes but prefer to use ConcurrentHashMap
Implementation	Buckets	Red-Black tree	HashTable and LinkedList using doubly linked list of buckets	Buckets
Comments	Efficient	Extra cost of maintaining TreeMap	Advantage of TreeMap without extra cost.	Obsolete

Difference between HashMap, HashTable ,ConcurrentHashMap

Property	HashMap	Hashtable	ConcurrentHashMap
Null values/keys	allowed	not allowed	not allowed
Is thread-safe	no	yes	yes
Lock mechanism	not applicable	locks the whole map	locks the portion of map
Iterator	fail-fast		fail-safe

Difference between HashMap and HashTable

HashMap	Hashtable
HashMap is **non synchronized**. It is **not-thread safe** and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is **thread-safe** and can be shared with many threads
HashMap **allows one null key and multiple null values.**	Hashtable **doesn't allow any null key or value.**
HashMap is a **new class introduced in JDK 1.2.**	Hashtable is a **legacy class**
HashMap is much faster and uses less memory than Hashtable **Unsynchronized objects are often much better in performance in compare to synchronized object**	Hashtable is **slow.**
HashMap is **traversed by Iterator.**	Hashtable is **traversed by Enumerator and Iterator.**
Iterator in HashMap is **fail-fast.**	Enumerator in Hashtable is **not fail-fast.**
HashMap inherits **AbstractMap** class.	Hashtable inherits **Dictionary** class.

Comparable vs Comparator

Comparable and Comparator in Java is used for sorting the collection of Objects.

Implementing Comparable means "I can compare myself with another object." This is typically useful when there's a single natural default comparison.

Implementing Comparator means "I can compare two other objects." This is typically useful when there are multiple ways of comparing two instances of a type – e.g. you could compare people by age, name etc.

java.lang.Comparable

To implement Comparable interface, class must implement a single method compareTo()
int a.compareTo(b)

You must modify the class whose instance you want to sort. So that only one sort sequence can be created per class.

java.util.Comparator

To implement Comparator interface, class must implement a single method compare()
int compare (a,b)

You build a class separate from class whose instance you want to sort. So that multiple sort sequence can be created per class.

Example for Comparable

```
import java.util.ArrayList;
import java.util.Collections;

public class ComparableExample {
    public static void main(String args[]) {
        ArrayList<Employee> empList = new ArrayList<Employee>();
        empList.add(new Employee(110, "Praveen", 34, 200000));
        empList.add(new Employee(101, "Prasad", 35, 100000));
        empList.add(new Employee(108, "Ravi", 34, 500000));
        empList.add(new Employee(112, "Prakash", 38, 700000));
        Collections.sort(empList);
        System.out.println(empList);
    }
}
class Employee implements Comparable<Employee> {
    private int empId;
    private String empName;
    private int empAge;
    private double empSal;
```

```
    public Employee(int empId, String empName, int empAge, double empS
al) {
        this.empId = empId;
        this.empName = empName;
        this.empAge = empAge;
        this.empSal = empSal;
    }
    public int getEmpId() {
        return empId;
    }
    public String getEmpName() {
        return empName;
    }
    public int getEmpAge() {
        return empAge;
    }
    public double getEmpSal() {
        return empSal;
    }
    @Override
    public int compareTo(Employee emp) {
        if (this.empId == emp.empId)
            return 0;
        else if (this.empId > emp.empId)
            return +1;
        else
            return -1;
    }
    @Override
```

```
    public String toString() {
        return "Employee [empId=" + empId + ", empName=" + empName + ", empAge=" + empAge + ", empSal=" + empSal + "]";
    }
}
```

Output

[Employee [empId=101, empName=Prasad, empAge=35, empSal=100000.0], Employee [empId=108, empName=Ravi, empAge=34, empSal=500000.0], Employee [empId=110, empName=Praveen, empAge=34, empSal=200000.0], Employee [empId=112, empName=Prakash, empAge=38, empSal=700000.0]]

Example for Comparator

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class ComparatorExample {
    public static void main(String args[]) {
        ArrayList<EmployeeUpdated> empList = new ArrayList<EmployeeUpdated>();
        empList.add(new EmployeeUpdated(110, "Praveen", 34, 200000));
        empList.add(new EmployeeUpdated(101, "Prasad", 35, 100000));
        empList.add(new EmployeeUpdated(108, "Ravi", 34, 500000));
        empList.add(new EmployeeUpdated(112, "Prakash", 38, 700000));
        Collections.sort(empList, EmployeeUpdated.SalaryComparator);
        System.out.println("SalaryComparator \n" + empList);
        Collections.sort(empList, EmployeeUpdated.AgeComparator);
        System.out.println("AgeComparator \n" + empList);
        Collections.sort(empList, EmployeeUpdated.NameComparator);
        System.out.println("NameComparator \n" + empList);
```

```java
    }
}
class EmployeeUpdated {
    private int empId;
    private String empName;
    private int empAge;
    private double empSal;
    public EmployeeUpdated(int empId, String empName, int empAge, doub
le empSal) {
        this.empId = empId;
        this.empName = empName;
        this.empAge = empAge;
        this.empSal = empSal;
    }
    public int getEmpId() {
        return empId;
    }
    public String getEmpName() {
        return empName;
    }
    public int getEmpAge() {
        return empAge;
    }
    public double getEmpSal() {
        return empSal;
    }
    /**
     * Comparator to sort employees list or array in order of Salary
     */
```


Linktree: https://linktr.ee/praveenoruganti
Email: praveenorugantitech@gmail.com

```java
    public static Comparator<EmployeeUpdated> SalaryComparator = new C
omparator<EmployeeUpdated>() {

        @Override

        public int compare(EmployeeUpdated e1, EmployeeUpdated e2) {

            return (int) (e1.getEmpSal() - e2.getEmpSal());

        }

    };

    /**

     * Comparator to sort employees list or array in order of Age

     */

    public static Comparator<EmployeeUpdated> AgeComparator = new Comp
arator<EmployeeUpdated>() {

        @Override

        public int compare(EmployeeUpdated e1, EmployeeUpdated e2) {

            return e1.getEmpAge() - e2.getEmpAge();

        }

    };

    /**

     * Comparator to sort employees list or array in order of Name

     */

    public static Comparator<EmployeeUpdated> NameComparator = new Com
parator<EmployeeUpdated>() {

        @Override

        public int compare(EmployeeUpdated e1, EmployeeUpdated e2) {

            return e1.getEmpName().compareTo(e2.getEmpName());

        }

    };

    @Override

    public String toString() {
```

```
        return "EmployeeUpdated [empId=" + empId + ", empName=" + empN
ame + ", empAge=" + empAge + ", empSal=" + empSal

                + "]";

    }

}
```

Output
SalaryComparator [EmployeeUpdated [empId=101, empName=Prasad, empAge=35, empSal=100000.0], EmployeeUpdated [empId=110, empName=Praveen, empAge=34, empSal=200000.0], EmployeeUpdated [empId=108, empName=Ravi, empAge=34, empSal=500000.0], EmployeeUpdated [empId=112, empName=Prakash, empAge=38, empSal=700000.0]]AgeComparator [EmployeeUpdated [empId=110, empName=Praveen, empAge=34, empSal=200000.0], EmployeeUpdated [empId=108, empName=Ravi, empAge=34, empSal=500000.0], EmployeeUpdated [empId=101, empName=Prasad, empAge=35, empSal=100000.0], EmployeeUpdated [empId=112, empName=Prakash, empAge=38, empSal=700000.0]]NameComparator [EmployeeUpdated [empId=112, empName=Prakash, empAge=38, empSal=700000.0], EmployeeUpdated [empId=101, empName=Prasad, empAge=35, empSal=100000.0], EmployeeUpdated [empId=110, empName=Praveen, empAge=34, empSal=200000.0], EmployeeUpdated [empId=108, empName=Ravi, empAge=34, empSal=500000.0]]

Example for Comparator Using Lambda

```
import java.util.ArrayList;

import java.util.Comparator;


public class ComparatorUsingLambda {

    public static void main(String args[]) {

        ArrayList<EmployeeUpdated> empList = new ArrayList<EmployeeUpd
ated>();

        empList.add(new EmployeeUpdated(110, "Praveen", 34, 200000));

        empList.add(new EmployeeUpdated(101, "Prasad", 35, 100000));

        empList.add(new EmployeeUpdated(108, "Ravi", 34, 500000));

        empList.add(new EmployeeUpdated(112, "Prakash", 38, 700000));

        empList.sort(new Comparator<EmployeeUpdated>() {
```

```java
            @Override
            public int compare(EmployeeUpdated e1, EmployeeUpdated e2)
{
                return (int) (e1.getEmpSal() - e2.getEmpSal());
            }
        });
        System.out.println("SalaryComparator");
        empList.forEach((employee) -> System.out.println(employee));
        empList.sort(new Comparator<EmployeeUpdated>() {
            @Override
            public int compare(EmployeeUpdated e1, EmployeeUpdated e2)
{
                return (int) (e1.getEmpAge() - e2.getEmpAge());
            }
        });
        System.out.println("AgeComparator");
        empList.forEach((employee) -> System.out.println(employee));
        empList.sort(new Comparator<EmployeeUpdated>() {
            @Override
            public int compare(EmployeeUpdated e1, EmployeeUpdated e2)
{
                return e1.getEmpName().compareTo(e2.getEmpName());
            }
        });
        System.out.println("NameComparator");
        empList.forEach((employee) -> System.out.println(employee));
    }
}
```

Output

SalaryComparatorEmployeeUpdated [empId=101, empName=Prasad, empAge=35, empSal=100000.0]EmployeeUpdated [empId=110, empName=Praveen, empAge=34, empSal=200000.0]EmployeeUpdated [empId=108, empName=Ravi, empAge=34, empSal=500000.0]EmployeeUpdated [empId=112, empName=Prakash, empAge=38, empSal=700000.0]AgeComparatorEmployeeUpdated [empId=110, empName=Praveen, empAge=34, empSal=200000.0]EmployeeUpdated [empId=108, empName=Ravi, empAge=34, empSal=500000.0]EmployeeUpdated [empId=101, empName=Prasad, empAge=35, empSal=100000.0]EmployeeUpdated [empId=112, empName=Prakash, empAge=38, empSal=700000.0]NameComparatorEmployeeUpdated [empId=112, empName=Prakash, empAge=38, empSal=700000.0]EmployeeUpdated [empId=101, empName=Prasad, empAge=35, empSal=100000.0]EmployeeUpdated [empId=110, empName=Praveen, empAge=34, empSal=200000.0]EmployeeUpdated [empId=108, empName=Ravi, empAge=34, empSal=500000.0]

You can checkout my other ebooks in
https://praveenorugantitech.github.io/praveenorugantitech-ebooks/