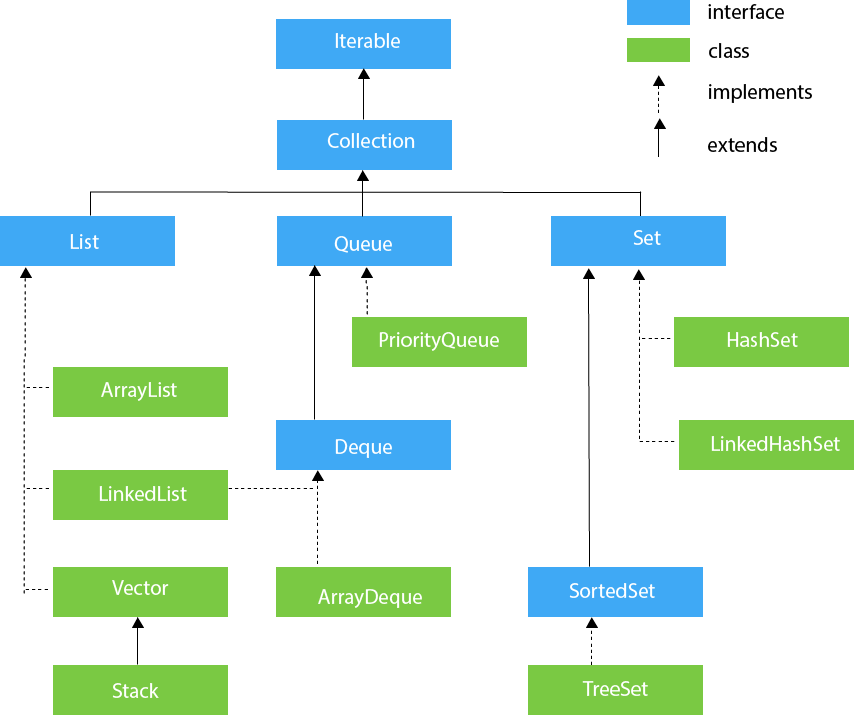
**Collections**

**Legacy Classes: Vector, Hashtable**

****

**Collection Chart**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Implements/**  **Extends** | **Order Maintain** | **Is Duplicate**  **Allow** | **Is Null**  **Allow?** | **Data**  **Structure** | **Synchronized** | **Methods** | **performance** |
| **List(I)** | Collection | Ordered Collection |  |  |  |  |  |  |
| **ArrayList** | List | Insertion | Yes |  | Dynamic Array | non synchronized. |  | Manipulation Slow  Because internally uses array.  ArrayList is initialized with a default capacity of 10.  an ArrayList is a resizable array |
| **LinkedList** | inherits the AbstractList class and implements List and Deque interfaces | Insertion order | Yes |  | Doubly linked list | non synchronized |  | Manipulation is fast Because internally uses a Doubly linked list |
| **Vector** | List | Insertion |  |  | Dynamic Array | Yes |  |  |
| **Stack** | Vector(Extends) | LIFO |  |  | LIFO |  | Push(),  peek() |  |
| **Queue(I)** |  | Ordered List |  | The NullPointerException is raised, if any null operation is done on the BlockingQueues. | FIFO |  |  | If it is required to have a thread safe implementation, PriorityBlockingQueue is an available option.  In Queue, insertion of elements occurs at the end of the list, and removal of elements occurs at the beginning of the list. |
| **Priority Queue** | Queue |  |  | No |  |  |  |  |
| **Deque(I)** | Queue |  |  |  |  |  |  | The Deque supports the addition as well as the removal of elements from both ends of the data structure. |
| **Array Deque** |  |  |  | Not Allow |  | Non Synchronized |  | Unlike Queue, we can add or remove elements from both sides.  ArrayDeque is faster than LinkedList and Stack. |
| **Set(I)** |  | Unordered Set | No,  Unique Only | Only One Null Value |  |  |  |  |
| **HashSet** | inherits the AbstractSet class and implements Set interface. | HashSet doesn't maintain the insertion order. Because inserted based on their hashcode. | No,  Unique Only | HashSet allows a null value. | HashTable(  the hashing mechanism used to store the elements.) | non-synchronized. |  | default capacity of HashSet is 16, and the load factor is 0.75.  the best approach for search operations.  HashSet implemented by hash table.  HashSet performs faster than TreeSet.  HashSet uses HashMap internally. |
| **LinkedHashSet** |  | Insertion Ordered | No,  unique elements only like HashSet. | doesn't allow null element. |  |  |  | Note: Keeping the insertion order in the LinkedHashset has some additional costs, both in terms of extra memory and extra CPU cycles. Therefore, if it is not required to maintain the insertion order, go for the lighter-weight HashMap or the HashSet instead. |
| **Sorted Set(I)/ Alternate of Set** | Set | Insertion Ordered |  |  |  |  |  |  |
| **TreeSet** | Sorted Set  (TreeSet class implements the NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.) | Ascending Order | TreeSet class contains unique elements only like HashSet. | doesn't allow null elements. |  | non-synchronized. |  | The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.  Java TreeSet class access and retrieval times are quiet fast.  TreeSet class access and retrieval times are quite fast.  TreeSet implemented by a Tree structure.  TreeSet uses TreeMap internally in java. |

|  |  |
| --- | --- |
| List(I) | List interface is the child interface of Collection interface  we can store the ordered collection of objects.  It can have duplicate values.  List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack. |
| ArrayList | The ArrayList class implements the List interface.  It uses a dynamic array to store the duplicate element of different data types.  The ArrayList class maintains the insertion order and is non-synchronized.  The elements stored in the ArrayList class can be randomly accessed. |
| LinkedList | LinkedList implements the Collection interface.  It uses a doubly linked list internally to store the elements.  It can store the duplicate elements. It maintains the insertion order and is not synchronized.  In LinkedList, the manipulation is fast because no shifting is required. |
| Vector | Vector uses a dynamic array to store the data elements.  It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework. |
| Stack | The stack is the subclass of Vector.  It implements the last-in-first-out data structure, i.e., Stack.  The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties. |
| Queue Interface | Queue interface maintains the first-in-first-out order.  It can be defined as an ordered list that is used to hold the elements which are about to be processed.  There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.  Queue interface can be instantiated as:  Queue<String> q1 = new PriorityQueue();  Queue<String> q2 = new ArrayDeque(); |
| PriorityQueue | The PriorityQueue class implements the Queue interface.  It holds the elements or objects which are to be processed by their priorities.  PriorityQueue doesn't allow null values to be stored in the queue. |
| Deque | Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.  Deque can be instantiated as:  Deque d = new ArrayDeque(); |
| ArrayDeque | ArrayDeque class implements the Deque interface.  It facilitates us to use the Deque.  Unlike queue, we can add or delete the elements from both the ends.  ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions. |
|  |  |
| Set Interface | Set Interface in Java is present in java.util package.  It extends the Collection interface.  It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set.  Set is implemented by HashSet, LinkedHashSet, and TreeSet. |
| HashSet | HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items. |
| LinkedHashSet | LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface.  Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements. |
| SortedSet | SortedSet is the alternate of Set interface that provides a total ordering on its elements.  The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.  SortedSet<data-type> set = **new** TreeSet(); |

**Q1: How HashMap works in Java?**

If anybody asks me to describe “**How HashMap works?** “, I simply answer: “**On principle of Hashing** “. As simple as it is. Now before answering it, one must be very sure to know at least basics of Hashing.

**Q2. What is Hashing?**

[Hashing](https://en.wikipedia.org/wiki/Hash_function) in its simplest form, is a way to assigning a unique code for any variable/object after applying any formula/algorithm on its properties.

A true [hash function](https://howtodoinjava.com/security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/) must follow this rule –

“Hash function should **return the same hash code each and every time** when the function is applied on same or equal objects. In other words, two equal objects must produce the same hash code consistently.”

**Note:** All objects in Java inherit a default implementation of hashCode() function defined in Object class. This function produces hash code by typically converting the internal address of the object into an integer, thus producing different hash codes for all different objects.

## Entry class in HashMap

static class Entry<K ,V> implements Map.Entry<K, V>

{

    final K key;

    V value;

    Entry<K ,V> next;

    final int hash;

    ...//More code goes here

}

**Java Comparable interface**

Java Comparable interface is used to order the objects of the user-defined class.

This interface is found in java.lang package and contains only one method named compareTo(Object).

It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only.

For example, it may be rollno, name, age or anything else.

**compareTo(Object obj) method:**

**public int compareTo(Object obj):** It is used to compare the current object with the specified object. It returns

1. positive integer, if the current object is greater than the specified object.

2. negative integer, if the current object is less than the specified object.

3. zero, if the current object is equal to the specified object.

However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

**We can sort the elements of:**

String objects

Wrapper class objects

User-defined class objects

**Collections class**

**Collections** class provides static methods for sorting the elements of collections. If collection elements are of Set or Map, we can use TreeSet or TreeMap. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

Method of Collections class for sorting List elements

**public void sort(List list):** It is used to sort the elements of List. List elements must be of the Comparable type.

Note: String class and Wrapper classes implement the Comparable interface by default. So if you store the objects of string or wrapper classes in a list, set or map, it will be Comparable by default.

**Java Comparator interface**

**Java Comparator interface** is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

**Methods of Java Comparator Interface**

public int compare(Object obj1, Object obj2): It compares the first object with the second object.

public boolean equals(Object obj): It is used to compare the current object with the specified object.

**NOTE: Always keep some definition in mind**

**1) What is the Collection framework in Java?**

Collection Framework is a combination of classes and interface, which is used to store and manipulate the data in the form of objects. It provides various classes such as ArrayList, Vector,? Stack, and HashSet, etc. and interfaces such as List, Queue, Set, etc. for this purpose.

**2) What are the main differences between array and collection(ArrayList)?**

Array and Collection are somewhat similar regarding storing the references of objects and manipulating the data, but they differ in many ways.

The main differences between the array and Collection are defined below:

* Arrays are always of fixed size, i.e., a user can not increase or decrease the length of the array according to their requirement or at runtime, but In Collection, size can be changed dynamically as per need.
* Arrays can only store homogeneous or similar type objects, but in Collection, heterogeneous objects can be stored.
* Arrays cannot provide the ?ready-made? methods for user requirements as sorting, searching, etc. but Collection includes readymade methods to use.

**3) Explain various interfaces used in Collection framework?**

Collection framework implements various interfaces, Collection interface and Map interface (java.util.Map) are the mainly used interfaces of Java Collection Framework. List of interfaces of Collection Framework is given below:

**1. Collection interface:** Collection (java.util.Collection) is the primary interface, and every collection must implement this interface.

**Syntax: public** **interface** Collection<E>**extends** Iterable

Where <E> represents that this interface is of Generic type

**2. List interface:**List interface extends the Collection interface, and it is an ordered collection of objects. It contains duplicate elements. It also allows random access of elements.

**Syntax: public** **interface** List<E> **extends** Collection<E>

**3. Set interface:** Set (java.util.Set) interface is a collection which cannot contain duplicate elements. It can only include inherited methods of Collection interface

**Syntax:** **public** **interface** Set<E> **extends** Collection<E>

**4. Queue interface:**Queue (java.util.Queue) interface defines queue data structure, which stores the elements in the form FIFO (first in first out).

**Syntax: public** **interface** Queue<E> **extends** Collection<E>

**5. Dequeue interface:** it is a double-ended-queue. It allows the insertion and removal of elements from both ends. It implants the properties of both Stack and queue so it can perform LIFO (Last in first out) stack and FIFO (first in first out) queue, operations.

**Syntax:** **public** **interface** Dequeue<E> **extends** Queue<E>

**5. Map interface:**A Map (java.util.Map) represents a key, value pair storage of elements. Map interface does not implement the Collection interface. It can only contain a unique key but can have duplicate elements. There are two interfaces which implement Map in java that are HashMap interface and Sorted Map.

**4) What is the difference between ArrayList and Vector?**

|  |  |  |
| --- | --- | --- |
| No. | ArrayList | Vector |
| 1) | ArrayList is not synchronized. | Vector is synchronized. |
| 2) | ArrayList is not a legacy class. | Vector is a legacy class. |
| 3) | ArrayList increases its size by 50% of the array size. | Vector increases its size by doubling the array size. |
| 4) | ArrayList is not ?thread-safe? as it is not synchronized. | Vector list is ?thread-safe? as it?s every method is synchronized. |

**5) What is the difference between ArrayList and LinkedList?**

|  |  |  |
| --- | --- | --- |
| No. | ArrayList | LinkedList |
| 1) | ArrayList uses a dynamic array. | LinkedList uses a doubly linked list. |
| 2) | Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3) | ArrayList is better to store and fetch data. | LinkedList is better to manipulate data. |
| 4) | ArrayList provides random access. | LinkedList does not provide random access. |
| 5) | ArrayList takes less memory overhead as it stores only object | LinkedList takes more memory overhead, as it stores the object as well as the address of that object. |
|  | An ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |

**6) What is the difference between Iterator and List Iterator?**

Iterator traverses the elements in the forward direction only whereas ListIterator traverses the elements into forward and backward direction.

|  |  |  |
| --- | --- | --- |
| No. | Iterator | ListIterator |
| 1) | The Iterator traverses the elements in the forward direction only. | ListIterator traverses the elements in backward and forward directions both. |
| 2) | The Iterator can be used in List, Set, and Queue. | ListIterator can be used in List only. |
| 3) | The Iterator can only perform remove operation while traversing the collection. | ListIterator can perform ?add,? ?remove,? and ?set? operation while traversing the collection. |

**7) What is the difference between Iterator and Enumeration?**

|  |  |  |
| --- | --- | --- |
| No. | Iterator | Enumeration |
| 1) | The Iterator can traverse legacy and non-legacy elements. | Enumeration can traverse only legacy elements. |
| 2) | The Iterator is fail-fast. | Enumeration is not fail-fast. |
| 3) | The Iterator is slower than Enumeration. | Enumeration is faster than Iterator. |
| 4) | The Iterator can perform remove operation while traversing the collection. | The Enumeration can perform only traverse operation on the collection. |
|  |  |  |

**8) What is the difference between List and Set?**

The List and Set both extend the collection interface. However, there are some differences between the both which are listed below.

The List can contain duplicate elements whereas Set includes unique items.

The List is an ordered collection which maintains the insertion order whereas Set is an unordered collection which does not preserve the insertion order.

The List interface contains a single legacy class which is Vector class whereas Set interface does not have any legacy class.

The List interface can allow n number of null values whereas Set interface only allows a single null value.

**9) What is the difference between HashSet and TreeSet?**

The HashSet and TreeSet, both classes, implement Set interface. The differences between the both are listed below.

HashSet maintains no order whereas TreeSet maintains ascending order.

HashSet implemented by hash table whereas TreeSet implemented by a Tree structure.

HashSet performs faster than TreeSet.

HashSet uses HashMap internally whereas TreeSet uses TreeMap internally in java.

Note: why tree set does not contain null value;

**Adding null values to a tree set**

TreeSet adds elements to it according to their natural order. This internally compares the elements with each other using the compareTo (or compare) method.

If you try to compare any object with a null value using one of these methods, a NullPointerException will be thrown.

Therefore, if you try to add null values to a TreeSet it generates a NullPointerException at the run time.

**10) What is the difference between Set and Map?**

The differences between the Set and Map are given below.

Set contains values only whereas Map contains key and values both.

Set contains unique values whereas Map can contain unique Keys with duplicate values.

Set holds a single number of null value whereas Map can include a single null key with n number of null values.

**11) What is the difference between HashSet and HashMap?**

The differences between the HashSet and HashMap are listed below.

HashSet contains only values whereas HashMap includes the entry (key, value). HashSet can be iterated, but HashMap needs to convert into Set to be iterated.

HashSet implements Set interface whereas HashMap implements the Map interface

HashSet cannot have any duplicate value whereas HashMap can contain duplicate values with unique keys.

HashSet contains the only single number of null value whereas HashMap can hold a single null key with n number of null values.

**12) What is the difference between HashMap and TreeMap?**

The differences between the HashMap and TreeMap are given below.

HashMap maintains no order, but TreeMap maintains ascending order.

HashMap is implemented by hash table whereas TreeMap is implemented by a Tree structure.

HashMap can be sorted by Key or value whereas TreeMap can be sorted by Key.

HashMap may contain a null key with multiple null values whereas TreeMap cannot hold a null key but can have multiple null values.

**13) What is the difference between HashMap and Hashtable?**

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

|  |  |
| --- | --- |
| HashMap | Hashtable |
| 1) 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. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

**14) What is the difference between Collection and Collections?**

The differences between the Collection and Collections are given below.

The Collection is an interface whereas Collections is a class.

The Collection interface provides the standard functionality of data structure to List, Set, and Queue. However, Collections class is to sort and synchronize the collection elements.

The Collection interface provides the methods that can be used for data structure whereas Collections class provides the static methods which can be used for various operation on a collection.

**15) What is the difference between Comparable and Comparator?**

|  |  |  |
| --- | --- | --- |
| **No.** | **Comparable** | **Comparator** |
| 1) | Comparable provides only one sort of sequence. | The Comparator provides multiple sorts of sequences. |
| 2) | It provides one method named compareTo(). | It provides one method named compare(). |
| 3) | It is found in java.lang package. | It is located in java.util package. |
| 4) | If we implement the Comparable interface, The actual class is modified. | The actual class is not changed. |

**Q.Difference between Hashmap and Concurrent Hashmap?**

[HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) is the Class which is under Traditional Collection and ConcurrentHashMap is a Class which is under Concurrent Collections, apart from this there are various differences between them which are:

* HashMap is non-Synchronized in nature i.e. HashMap is not Thread-safe whereas ConcurrentHashMap is Thread-safe in nature.
* HashMap performance is relatively high because it is non-synchronized in nature and any number of threads can perform simultaneously. But ConcurrentHashMap performance is low sometimes because sometimes Threads are required to wait on ConcurrentHashMap.
* While one thread is Iterating the HashMap object, if other thread try to add/modify the contents of Object then we will get Run-time exception saying **ConcurrentModificationException**.Whereas In ConcurrentHashMap we wont get any exception while performing any modification at the time of Iteration.

**16) What do you understand by BlockingQueue?**

BlockingQueue is an interface which extends the Queue interface.

It provides concurrency in the operations like retrieval, insertion, deletion.

While retrieval of any element, it waits for the queue to be non-empty. While storing the elements, it waits for the available space.

BlockingQueue cannot contain null elements, and implementation of BlockingQueue is thread-safe.

**Syntax: public** **interface** BlockingQueue<E> **extends** Queue <E>

Note: Java BlockingQueue interface is part of java collections framework and it’s primarily used for implementing producer consumer problem. We don’t need to worry about waiting for the space to be available for producer or object to be available for consumer in BlockingQueue because it’s handled by implementation classes of BlockingQueue.

Java provides several BlockingQueue implementations such as ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue etc.

**17) What is the advantage of Properties file?**

If you change the value in the properties file, you don't need to recompile the java class. So, it makes the application easy to manage. It is used to store information which is to be changed frequently. Consider the following example.

**import** java.util.\*;  **import** java.io.\*;

**public** **class** Test {

**public** **static** **void** main(String[] args)**throws** Exception{

    FileReader reader=**new** FileReader("db.properties");

       Properties p=**new** Properties();

    p.load(reader);

       System.out.println(p.getProperty("user"));

    System.out.println(p.getProperty("password"));

}  }  **Output:** system oracle

**18) What does the hashCode() method?**

The hashCode() method returns a hash code value (an integer number).

The hashCode() method returns the same integer number if two keys (by calling equals() method) are identical.

However, it is possible that two hash code numbers can have different or the same keys.

If two objects do not produce an equal result by using the equals() method, then the hashcode() method will provide the different integer result for both the objects.

**19) Why we override equals() method?**

The equals method is used to check whether two objects are the same or not. It needs to be overridden if we want to check the objects based on the property.

For example, Employee is a class that has 3 data members: id, name, and salary. However, we want to check the equality of employee object by the salary. Then, we need to override the equals() method.

**20) How to synchronize List, Set and Map elements?**

Yes, Collections class provides methods to make List, Set or Map elements as synchronized:

|  |
| --- |
| public static List synchronizedList(List l){} |
| public static Set synchronizedSet(Set s){} |
| public static SortedSet synchronizedSortedSet(SortedSet s){} |
| public static Map synchronizedMap(Map m){} |
| public static SortedMap synchronizedSortedMap(SortedMap m){} |

**21) What is the advantage of the generic collection?**

There are three main advantages of using the generic collection.

If we use the generic class, we don't need typecasting.

It is type-safe and checked at compile time.

Generic confirms the stability of the code by making it bug detectable at compile time.

**22) What is hash-collision in Hashtable and how it is handled in Java?**

Two different keys with the same hash value are known as hash-collision. Two separate entries will be kept in a single hash bucket to avoid the collision. There are two ways to avoid hash-collision.

Separate Chaining

Open Addressing

**23) What is the Dictionary class?**

The Dictionary class provides the capability to store key-value pairs.

**24) What is the default size of load factor in hashing based collection?**

The default size of load factor is **0.75**. The default capacity is computed as initial capacity \* load factor. For example, 16 \* 0.75 = 12. So, 12 is the default capacity of Map.

**25) What do you understand by fail-fast?**

The Iterator in java which immediately throws ConcurrentmodificationException, if any structural modification occurs in, is called as a Fail-fast iterator. Fail-fats iterator does not require any extra space in memory.

**26)**[**What is the difference between Array and ArrayList?**](https://www.javatpoint.com/array-vs-arraylist-in-java)

The main differences between the Array and ArrayList are given below.

|  |  |  |
| --- | --- | --- |
| **SN** | **Array** | **ArrayList** |
| 1 | The Array is of fixed size, means we cannot resize the array as per need. | ArrayList is not of the fixed size we can change the size dynamically. |
| 2 | Arrays are of the static type. | ArrayList is of dynamic size. |
| 3 | Arrays can store primitive data types as well as objects. | ArrayList cannot store the primitive data types it can only store the objects. |

**27)**[**What is the difference between the length of an Array and size of ArrayList?**](https://www.javatpoint.com/difference-between-length-of-array-and-size-of-arraylist-in-java)

The length of an array can be obtained using the property of length whereas ArrayList does not support length property, but we can use size() method to get the number of objects in the list.

**Finding the length of the array**

Int [] array = **new** **int**[4];

System.out.println("The size of the array is " + array.length);

**Finding the size of the ArrayList**

ArrayList<String> list=**new** ArrayList<String>();

list.add("ankit");

list.add("nippun");

System.out.println(list.size());

**28)**[**How to convert ArrayList to Array and Array to ArrayList?**](https://www.javatpoint.com/how-to-convert-arraylist-to-array-and-array-to-arraylist-in-java)

We can convert an Array to ArrayList by using the asList() method of Arrays class. asList() method is the static method of Arrays class and accepts the List object. Consider the following syntax:

Arrays.asList(item)

We can convert an ArrayList to Array using toArray() method of the ArrayList class. Consider the following syntax to convert the ArrayList to the List object.

List\_object.toArray(**new** String[List\_object.size()])

**29)**[**How to make Java ArrayList Read-Only?**](https://www.javatpoint.com/how-to-make-java-arraylist-read-only)

We can obtain java ArrayList Read-only by calling the Collections.unmodifiableCollection() method. When we define an ArrayList as Read-only then we cannot perform any modification in the collection through  add(), remove() or set() method.

List<String>fruitList = new ArrayList<String>();

fruitList.add("Mango");

List<String>unmodifiableList= Collections.unmodifiableList(fruitList);

unmodifiableList.add("INDIA");

System.out.println(fruitList);

**30)**[**How to remove duplicates from ArrayList?**](https://www.javatpoint.com/how-to-remove-duplicates-from-arraylist-in-java)

There are two ways to remove duplicates from the ArrayList.

**Using HashSet:** By using HashSet we can remove the duplicate element from the ArrayList, but it will not then preserve the insertion order.

**Using LinkedHashSet:** We can also maintain the insertion order by using LinkedHashSet instead of HashSet.

The Process to remove duplicate elements from ArrayList using the LinkedHashSet:

Copy all the elements of ArrayList to LinkedHashSet.

Empty the ArrayList using clear() method, which will remove all the elements from the list.

Now copy all the elements of LinkedHashset to ArrayList.

**31)**[**How to reverse ArrayList?**](https://www.javatpoint.com/how-to-reverse-arraylist-in-java)

To reverse an ArrayList, we can use reverse() method of Collections class. Consider the following example.

**import** java.util.ArrayList;  **import** java.util.Collection;  **import** java.util.Collections;  **import** java.util.Iterator;

**import** java.util.List;

**public** **class** ReverseArrayList {

**public** **static** **void** main(String[] args) {

     List list = **new** ArrayList<>();

     list.add(10);       list.add(50);       list.add(30);

     Iterator i = list.iterator();       System.out.println("printing the list....");

**while**(i.hasNext())

     {           System.out.println(i.next());       }

     Iterator i2 = list.iterator();

     Collections.reverse(list);       System.out.println("printing list in reverse order....");

**while**(i2.hasNext())

     {

         System.out.println(i2.next());

     }

    }

}

**32)**[**How to sort ArrayList in descending order?**](https://www.javatpoint.com/how-to-sort-java-arraylist-in-descending-order)

To sort the ArrayList in descending order, we can use the reverseOrder method of Collections class. Consider the following example.

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.Collections;

**import** java.util.Comparator;

**import** java.util.Iterator;

**import** java.util.List;

**public** **class** ReverseArrayList {

**public** **static** **void** main(String[] args) {

     List list = **new** ArrayList<>();

     list.add(10);       list.add(50);       list.add(30);       list.add(60);       list.add(20);       list.add(90);

     Iterator i = list.iterator();

     System.out.println("printing the list....");

**while**(i.hasNext())

     {

         System.out.println(i.next());

     }

    Comparator cmp = Collections.reverseOrder();

    Collections.sort(list,cmp);

     System.out.println("printing list in descending order....");

     Iterator i2 = list.iterator();

**while**(i2.hasNext())

     {

         System.out.println(i2.next());

     }

}

}

**33)**[**How to synchronize ArrayList?**](https://www.javatpoint.com/how-to-synchronize-arraylist-in-java)

We can synchronize ArrayList in two ways.

Using Collections.synchronizedList() method

Using CopyOnWriteArrayList<T>

**34)**[**When to use ArrayList and LinkedList?**](https://www.javatpoint.com/when-to-use-arraylist-and-linkedlist-in-java)

LinkedLists are better to use for the update operations whereas ArrayLists are better to use for the search operations.

========================================= After Print =================================================

**All In Details**

**LinkedList:**

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

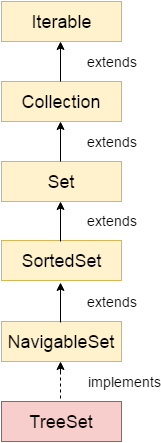
The important points about Java LinkedList are:

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.
* Java LinkedList class can be used as a list, stack or queue.

### Hierarchy of LinkedList class

As shown in the above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

# Java TreeSet class



Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

* Java TreeSet class contains unique elements only like HashSet.
* Java TreeSet class access and retrieval times are quiet fast.
* Java TreeSet class doesn't allow null element.
* Java TreeSet class is non synchronized.
* Java TreeSet class maintains ascending order.

### Hierarchy of TreeSet class

As shown in the above diagram, Java TreeSet class implements the NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

TreeSet<Integer> set=**new** TreeSet<Integer>();

set.add("Ajay");

System.out.println("Traversing element through Iterator in descending order");

Iterator i=set.descendingIterator();

**while**(i.hasNext())

{

System.out.println(i.next());

}

System.out.println("Highest Value: "+set.pollFirst());

System.out.println("Lowest Value: "+set.pollLast());

**Note:** TreeSet example where we are adding books to set and printing all the books. The elements in TreeSet must be of a Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in TreeSet, you need to implement the Comparable interface.

# Java EnumSet class

Java EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.

**EnumSet class hierarchy**

The hierarchy of EnumSet class is given in the figure given below.

**EnumSet class declaration**

Let's see the declaration for java.util.EnumSet class.

**public** **abstract** **class** EnumSet<E **extends** Enum<E>> **extends** AbstractSet<E> **implements** Cloneable, Serializable

**import** java.util.\*;

**enum** days {

  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

}

**public** **class** EnumSetExample {

**public** **static** **void** main(String[] args) {

    Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);

    // Traversing elements

    Iterator<days> iter = set.iterator();

**while** (iter.hasNext())

      System.out.println(iter.next());

Set<days> set1 = EnumSet.allOf(days.**class**);

      System.out.println("Week Days:"+set1);

      Set<days> set2 = EnumSet.noneOf(days.**class**);

      System.out.println("Week Days:"+set2);

  }

}

o/p:

TUESDAY

WEDNESDAY

Week Days:[SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY]

Week Days:[]

**Interview Question**

# ConcurrentHashMap in java:

# **Prerequisites:**Need of ConcurrentMap **ConcurrentHashMap** ConcurrentHashMap class is introduced in JDK 1.5, which implements ConcurrentMap as well as Serializable interface also. ConcureentHashMap is enhancement of HashMap as we know that while dealing with Threads in our application HashMap is not a good choice because performance wise HashMap is not upto the mark.

**Key points of ConcurrentHashMap:**

* The underlined data structure for ConcurrentHashMap is Hashtable.
* ConcurrentHashMap class is thread-safe i.e. multiple thread can operate on a single object without any complications.
* At a time any number of threads are applicable for read operation without locking the ConcurrentHashMap object which is not there in HashMap.
* In ConcurrentHashMap, the Object is divided into number of segments according to the concurrency level.
* Default concurrency-level of ConcurrentHashMap is 16.
* In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updation in object, thread must lock the particular segment in which thread want to operate.This type of locking mechanism is known as **Segment locking or bucket locking**.Hence at a time 16 updation operations can be performed by threads.
* null insertion is not possible in ConcurrentHashMap as key or value.

**Constructors of ConcurrentHashMap:**

1. **ConcurrentHashMap m=new ConcurrentHashMap();**:Creates a new, empty map with a default initial capacity (16), load factor (0.75) and concurrencyLevel (16).
2. **ConcurrentHashMap m=new ConcurrentHashMap(int initialCapacity);**:Creates a new, empty map with the specified initial capacity, and with default load factor (0.75) and concurrencyLevel (16).
3. **ConcurrentHashMap m=new ConcurrentHashMap(int initialCapacity, float loadFactor);**:  
   Creates a new, empty map with the specified initial capacity and load factor and with the default concurrencyLevel (16).
4. **ConcurrentHashMap m=new ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel);**:Creates a new, empty map with the specified initial capacity, load factor and concurrency level.
5. **ConcurrentHashMap m=new ConcurrentHashMap(Map m);**:Creates a new map with the same mappings as the given map.

**// Java program to demonstrate working of ConcurrentHashMap**

**import java.util.concurrent.\*;**

**class ConcurrentHashMapDemo {**

**public static void main(String[] args)**

**{**

**ConcurrentHashMap m = new ConcurrentHashMap();**

**m.put(100, "Hello");**

**m.put(101, "Geeks");**

**m.put(102, "Geeks");**

**// Here we cant add Hello because 101 key**

**// is already present in ConcurrentHashMap object**

**m.putIfAbsent(101, "Hello");**

**// We can remove entry because 101 key**

**// is associated with For value**

**m.remove(101, "Geeks");**

**// Now we can add Hello**

**m.putIfAbsent(103, "Hello");**

**// We cant replace Hello with For**

**m.replace(101, "Hello", "For");**

**System.out.println(m);**

**}**

**}**

### Ways to iterate the elements of the collection in Java(https://www.javatpoint.com/java-arraylist)

There are various ways to traverse the collection elements:

1. By Iterator interface.
2. By for-each loop.
3. By ListIterator interface.
4. By for loop.
5. By forEach() method.
6. By forEachRemaining() method.

**import** java.util.\*;

**class** ArrayList4{

**public** **static** **void** main(String args[]){

    ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist

           list.add("Ravi");//Adding object in arraylist

           list.add("Vijay");

           list.add("Ravi");

           list.add("Ajay");

           System.out.println("Traversing list through List Iterator:");

           //Here, element iterates in reverse order

              ListIterator<String> list1=list.listIterator(list.size());

**while**(list1.hasPrevious())

              {

                  String str=list1.previous();

                  System.out.println(str);

              }

        System.out.println("Traversing list through for loop:");

**for**(**int** i=0;i<list.size();i++)

           {

            System.out.println(list.get(i));

           }

        System.out.println("Traversing list through forEach() method:");

        //The forEach() method is a new feature, introduced in Java 8.

            list.forEach(a->{ //Here, we are using lambda expression

                System.out.println(a);

              });

            System.out.println("Traversing list through forEachRemaining() method:");

              Iterator<String> itr=list.iterator();

              itr.forEachRemaining(a-> //Here, we are using lambda expression

              {

            System.out.println(a);

              });

 }

}

**New**

**How to Compare Two ArrayList in Java**

**There are following ways to compare two ArrayList in Java:**

**(https://www.javatpoint.com/how-to-compare-two-arraylist-in-java)**

* Java equals() method
* Java removeAll() method
* Java retainAll() method
* Java ArrayList.contains() method
* Java contentEquals() method
* Java Stream interface

**Java retainAll() method (returns common elements in both lists)**

**Example:**

ArrayList<String> firstList=**new** ArrayList<String>(Arrays.asList("M", "W", "J", "K", "T"));

List<String> secondList=**new** ArrayList<String>(Arrays.asList("M", "W", "E", "K", "T"));

secondList.retainAll(firstList);

System.out.println(secondList);

Output: [M, W, K, T] // common elements

OR

ArrayList<String> firstList = new ArrayList<String>(Arrays.asList("Java", "Python", "Ruby", "Go"));

ArrayList<String> secondList = new ArrayList<String>(Arrays.asList("Java", "Python", "Ruby", "Go", "Perl"));

// Finds common elements

System.out.print("Common elements: " +firstList.stream().filter(secondList::contains).collect(Collectors.toList()));

Output: Common elements: [Java, Python, Ruby, Go]

**Q: How to remove duplicates from ArrayList in Java?**

public class RemoveDuplicateArrayList {

public static void main(String[] args) {

List<String> list = new ArrayList<String>();

l.add("Mango");

l.add("Banana");

l.add("Mango");

l.add("Apple");

System.out.println(l.toString());

Set<String> s = new LinkedHashSet<String>(list);

System.out.println(s);

}

}

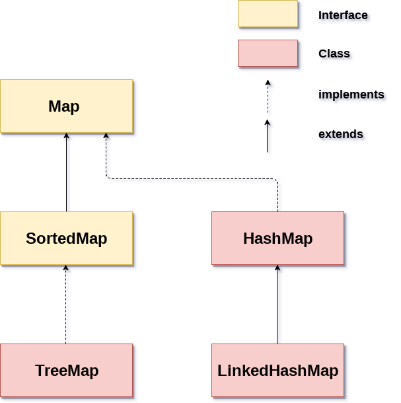
# Java Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

## Java Map Hierarchy

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:



A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method.

|  |  |
| --- | --- |
| HashMap | HashMap is the implementation of Map, but it doesn't maintain any order. |
| LinkedHashMap | LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order. |
| TreeMap | TreeMap is the implementation of Map and SortedMap. It maintains ascending order. |

Ques: How to Sort HashMap in Java.( <https://www.javatpoint.com/how-to-sort-hashmap-in-java>)

There are following ways to sort HashMap by keys:

* By using TreeMap
* By using LinkedHashMap

When we use LinkedHashMap, we should follow the process:

When we use LinkedHashMap, then we need to get Key set. Convert the Set into List, sort the list and then add the sorted list into LinkedHashMap in the same order. The same process we have done in the example Sort HashMap by Value.

**Q: Internal working of ArrayList or How add(Object) method works internally in ArrayList in Java.**

*ArrayList****internally uses  array******object****to add(or store) the elements. In other words, ArrayList is backed by Array data -structure.The array of ArrayList is****resizable (or dynamic).***

**Q: Internal working of Hashset?**

**Difference Between Concurrent Hashmap Vs Synchronized Hashmap**

[**https://www.naukri.com/code360/library/difference-between-concurrent-hashmap-vs-synchronized-hashmap**](https://www.naukri.com/code360/library/difference-between-concurrent-hashmap-vs-synchronized-hashmap)

**Introduction**

**This blog will discuss the differences between Concurrent Hashmap and Synchronized Hashmap or concurrent hashmap vs synchronized hashmap. But before we learn about these two hashmaps, let us understand the hashmap in general.**

**A Haspmap is one data structure that allows the programmer to store the data in a set of key and value pairs. Key value pairs mean we can access a particular value in a Hashmap with the help of its assigned key.**

**Example**

|  |  |
| --- | --- |
| **KEY** | **VALUE** |
| **12** | **“Car”** |
| **32** | **“Bike”** |
| **13** | **“Cycle”** |

**Hashmap**

This block is an example of a hashmap, and with the help of a key, you can access a particular value like key = 12 has value car mapped to it.

**ConcurrentHashMap**

ConcurrentHashMap is part of the Java Collections Framework introduced in Java 5. It provides thread-safe access to a hash map, allowing multiple threads to read and write concurrently without external synchronization. It achieves this concurrency by dividing the map into segments, and each segment is managed independently with its own lock. This means that multiple threads can access different segments of the map concurrently, improving concurrency and scalability.

**Synchronized HashMap**

A synchronized HashMap is a regular HashMap wrapped with synchronization mechanisms to make it thread-safe. This is typically achieved by using external synchronization, such as the synchronized keyword or using locks. Every method that modifies the map (e.g., put, remove, etc.) is synchronized, ensuring that only one thread can modify the map at a time.

**Need for Concurrent and Synchronized Hashmap**

Even though we have a hashmap, why do we still need a concurrent and synchronized hashmap?

Normal hashmap is ineffective when working in a concurrent multithreaded environment, and thread safety is a priority. Concurrency means running more than one program or application in parallel with the help of threads. Threads are a light weighted processes in a process.

The alternative to this problem was a concurrent and synchronized hashmap. We can work in a concurrent multithreading environment with the help of concurrent hashmap and synchronized hashmap.

Now let's talk about concurrent hashmap vs synchronized hashmap, which is our main focus for this blog. We will discuss some points to clarify the concurrent hashmap vs synchronized hashmap. Also, we implement both of the classes in a java program to have a proper understanding of concurrent hashmap vs synchronized hashmap.

**Concurrent Hashmap vs Synchronized Hashmap**

| **Concurrent Hashmap** | **Synchronized Hashmap** |
| --- | --- |
| **We need to include the java.util.concurrent to implement the concurrent hashmap interface.\* class in java.** | **To implement a synchronized hashmap, we need to include the method available in the collection class, java.util.Collections.synchronizedMap;** |
| **We cannot add null key-value pair in the concurrent hashmap.** | **We can add null key-value pair in the synchronized hashmap.** |
| **We can perform the read and write operation concurrently in a concurrent hashmap that performs better.** | **Concurrent behavior is not available in the synchronized hashmap completely.** |
| **Concurrent hashmap is scalable.** | **Synchronized hashmap is not scalable.** |
| **Thread safety in concurrent hashmap is not applied on the whole object, but at the bucket, a level called fragments.** | **Synchronized hashmap is thread-safe and is applied on a whole object.** |

The above points cover the concurrent hashmap vs synchronized hashmap. Now let's discuss the implementation of concurrent and synchronized hashmap so you can better understand the topic of concurrent hashmap vs synchronized hashmap.

**Implementation**

* **Java**

**import java.util.Collections;**

**import java.util.HashMap;**

**import java.util.Map;**

**import java.util.concurrent.ConcurrentHashMap;**

**public class Main**

**{**

**public static void main(String[] args) {**

**/\* Initializing a synchronized Hashmap interface\*/**

**Map<Integer,String> syncMap = Collections.synchronizedMap(new HashMap<Integer, String>());**

**/\*Try catch block demonstrate the synchronized hashmap \*/**

**try {**

**syncMap.put(null,null);**

**} catch(Exception e) {**

**System.out.println("error in syncMap");**

**}**

**/\* Initializing a concurrent Hashmap interface\*/**

**ConcurrentHashMap<Integer, String> concMap = new ConcurrentHashMap<>();**

**try {**

**concMap.put(null,null);**

**} catch(Exception e) {**

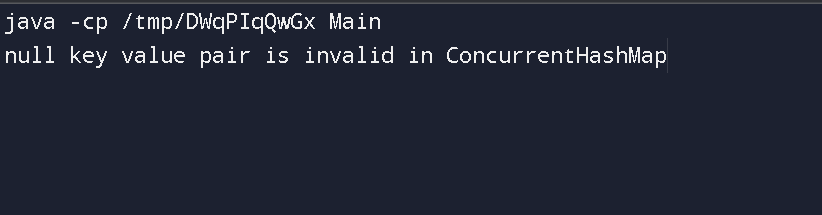
**System.out.println("null key-value pair is invalid in ConcurrentHashMap");**

**}**

**}**

**}**

[**Run Code**](https://www.naukri.com/code360/online-compiler/online-java-compiler)

****

**The above programs show how we can implement both concurrent and synchronized hashmaps. As mentioned in the blog, to implement a synchronized hashmap, you need to import the collection class in java and the concurrent class to implement a concurrent hashmap.**

**This program shows that we can use null key-value pair in a synchronized hashmap but not in a concurrent hashmap which is one of the differences in concurrent hashmap vs synchronized hashmap.**

**With all the mentioned points and content, we hope you have understood the concurrent hashmap vs synchronized hashmap means the difference between them.**