# Collections

* **It i**s a framework that provides an architecture to store and manipulate the group of objects.
* Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
* Java Collection means a single unit of objects.
* Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

#### What is Collection in Java

A Collection represents a single unit of objects, i.e., a group.

#### What is a framework in Java

* It provides readymade architecture.
* It represents a set of classes and interfaces.
* It is optional.

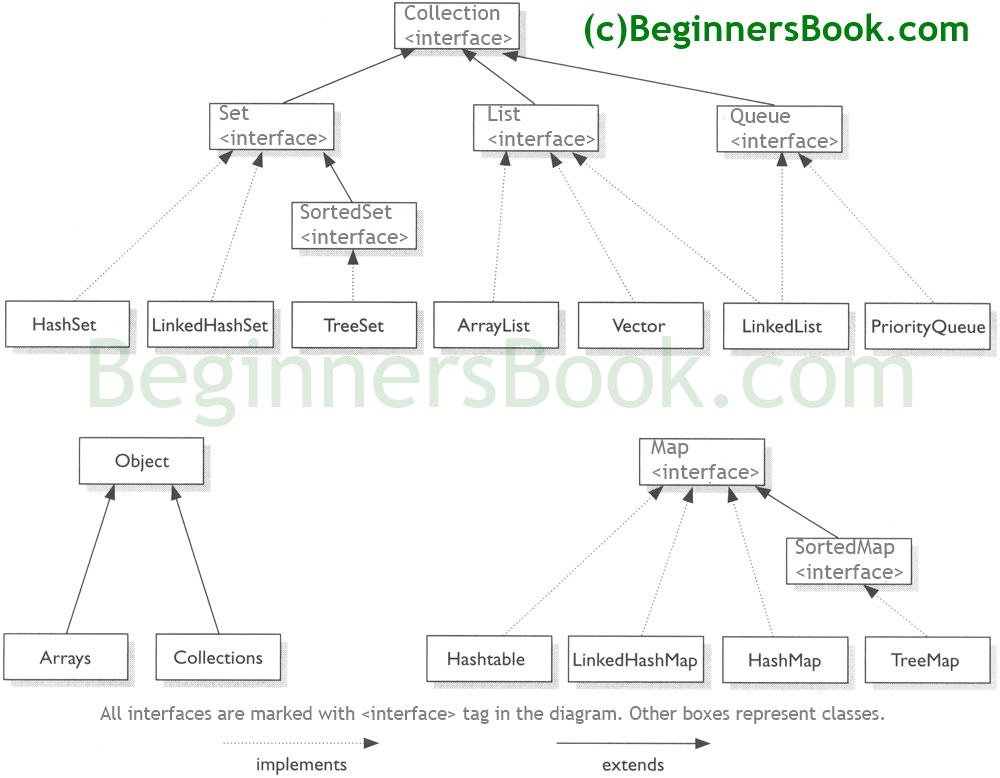
#### What is Collection framework

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

### Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.



### Iterator interface

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in a forward direction only. |

#### Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean **hasNext()** | It returns true if the iterator has more elements otherwise it returns false. |
| 2 | public Object **next()** | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void **remove()** | It removes the last elements returned by the iterator. It is less used. |

## **Iterable Interface**

## **Root interface** for all the collection classes.

## The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

## It contains only one **abstract method**. i.e.,

Iterator<T> iterator()

It returns the iterator over the elements of type T.

## **Collection Interface**

* interface which is **implemented by all the classes** in the collection framework.
* In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

## **List Interface**

* **child interface of Collection interface**.
* It inhibits a list type data structure in which 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.

To instantiate the List interface, we must use :

List <data-type> list1= **new** ArrayList();

List <data-type> list2 = **new** LinkedList();

List <data-type> list3 = **new** Vector();

List <data-type> list4 = **new** Stack();

* There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

## **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();

## **Set Interface**

* 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.

Set can be instantiated as:

Set<data-type> s1 = **new** HashSet<data-type>();

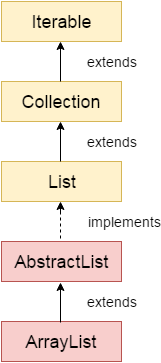
Set<data-type> s2 = **new** LinkedHashSet<data-type>();

Set<data-type> s3 = **new** TreeSet<data-type>();

**ArrayList**

* Java ArrayList class uses a dynamic array for storing the elements.
* It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

* Java ArrayList class can contain **duplicate elements**.
* Java ArrayList class maintains **insertion order**.
* Java ArrayList class is **non synchronized**.
* Java ArrayList allows **random access** because array works at the **index basis**.
* In Java ArrayList class, **manipulation is slow** because a lot of shifting needs to occur if any element is removed from the array list.

Consider the following example.

**import** java.util.\*;

**class** TestJavaCollection1{

**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");

//Traversing list through Iterator

Iterator itr=list.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Ravi

Vijay

Ravi

Ajay

## Methods of ArrayList class

In the above example we have used methods such as add() and remove(). However there are number of methods available which can be used directly using object of ArrayList class. Let’s discuss few important methods of ArrayList class.

1) **add( Object o)**: This method adds an object o to the arraylist.

obj.add("hello");

This statement would add a string hello in the arraylist at last position.

2) **add(int index, Object o)**: It adds the object o to the array list at the given index.

obj.add(2, "bye");

It will add the string bye to the 2nd index (3rd position as the array list starts with index 0) of array list.

3) **remove(Object o)**: Removes the object o from the ArrayList.

obj.remove("Chaitanya");

This statement will remove the string “Chaitanya” from the ArrayList.

4) **remove(int index)**: Removes element from a given index.

obj.remove(3);

It would remove the element of index 3 (4th element of the list – List starts with o).

5) **set(int index, Object o)**: Used for updating an element. It replaces the element present at the specified index with the object o.

obj.set(2, "Tom");

It would replace the 3rd element (index =2 is 3rd element) with the value Tom.

6)**int indexOf(Object o)**: Gives the index of the object o. If the element is not found in the list then this method returns the value -1.

int pos = obj.indexOf("Tom");

This would give the index (position) of the string Tom in the list.

7) **Object get(int index)**: It returns the object of list which is present at the specified index.

String str= obj.get(2);

Function get would return the string stored at 3rd position (index 2) and would be assigned to the string “str”. We have stored the returned value in string variable because in our example we have defined the ArrayList is of String type. If you are having integer array list then the returned value should be stored in an integer variable.

8) **int size()**: It gives the size of the ArrayList – Number of elements of the list.

int numberofitems = obj.size();

9) **boolean contains(Object o)**: It checks whether the given object o is present in the array list if its there then it returns true else it returns false.

obj.contains("Steve");

It would return true if the string “Steve” is present in the list else we would get false.

10) **clear():** It is used for removing all the elements of the array list in one go. The below code will remove all the elements of ArrayList whose object is obj.

obj.clear();

## **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:

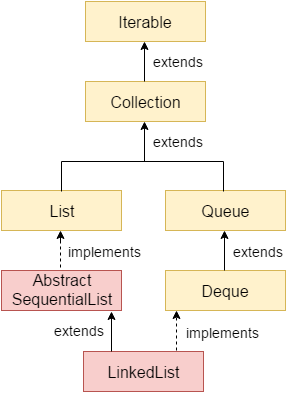
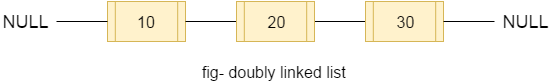
* Java LinkedList class can contain **duplicate elements**.
* Java LinkedList class maintains **insertion order**.
* Java LinkedList class is **non synchronized**.
* In Java LinkedList class, **manipulation is fast** because no shifting needs to occur.
* Java LinkedList class can be used as a **list,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

### Doubly Linked List

In the case of a doubly linked list, we can add or remove elements from both sides.

****

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection2{

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

LinkedList<String> al=**new** LinkedList<String>();

al.add("Ravi");

al.add("Vijay");

al.add("Ravi");

al.add("Ajay");

Iterator<String> itr=al.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Ravi

Vijay

Ravi

Ajay

## Methods of LinkedList class:

For all the examples in the below methods, consider llistobj as a reference for LinkedList<String>.

LinkedList<String> llistobj  = new LinkedList<String>();

1) **boolean add(Object item)**: It adds the item at the end of the list.

llistobj.add("Hello");

It would add the string “Hello” at the end of the linked list.

2) **void add(int index, Object item)**: It adds an item at the given index of the the list.

llistobj.add(2, "bye");

This will add the string “bye” at the 3rd position( 2 index is 3rd position as index starts with 0).

3) **boolean addAll(Collection c)**: It adds all the elements of the specified collection c to the list. It throws NullPointerException if the specified collection is null. Consider the below example –

LinkedList<String> llistobj = new LinkedList<String>();

ArrayList<String> arraylist= new ArrayList<String>();

arraylist.add("String1");

arraylist.add("String2");

llistobj.addAll(arraylist);

This piece of code would add all the elements of ArrayList to the LinkedList.

4) **boolean addAll(int index, Collection c)**: It adds all the elements of collection c to the list starting from a give index in the list. It throws NullPointerException if the collection c is null and IndexOutOfBoundsException when the specified index is out of the range.

llistobj.add(5, arraylist);

It would add all the elements of the ArrayList to the LinkedList starting from position 6 (index 5).

5) **void addFirst(Object item)**: It adds the item (or element) at the first position in the list.

llistobj.addFirst("text");

It would add the string “text” at the beginning of the list.

6) **void addLast(Object item)**: It inserts the specified item at the end of the list.

llistobj.addLast("Chaitanya");

This statement will add a string “Chaitanya” at the end position of the linked list.

7) **void clear()**: It removes all the elements of a list.

llistobj.clear();

8) **Object clone()**: It returns the copy of the list.

For e.g. My linkedList has four items: text1, text2, text3 and text4.

Object str= llistobj.clone();

System.out.println(str);

Output: The output of above code would be:

[text1, text2, text3, text4]

9) **boolean contains(Object item)**: It checks whether the given item is present in the list or not. If the item is present then it returns true else false.

boolean var = llistobj.contains("TestString");

It will check whether the string “TestString” exist in the list or not.

10) **Object get(int index)**: It returns the item of the specified index from the list.

Object var = llistobj.get(2);

It will fetch the 3rd item from the list.

11) **Object getFirst()**: It fetches the first item from the list.

Object var = llistobj.getFirst();

12) **Object getLast()**: It fetches the last item from the list.

Object var= llistobj.getLast();

13) **int indexOf(Object item)**: It returns the index of the specified item.

llistobj.indexOf("bye");

14) **int lastIndexOf(Object item)**: It returns the index of last occurrence of the specified element.

int pos = llistobj.lastIndexOf("hello);

integer variable pos will be having the index of last occurrence of string “hello”.

15) **Object poll()**: It returns and removes the first item of the list.

Object o = llistobj.poll();

16) **Object pollFirst()**: same as poll() method. Removes the first item of the list.

Object o = llistobj.pollFirst();

17) **Object pollLast()**: It returns and removes the last element of the list.

Object o = llistobj.pollLast();

18) **Object remove()**: It removes the first element of the list.

llistobj.remove();

19) **Object remove(int index)**: It removes the item from the list which is present at the specified index.

llistobj.remove(4);

It will remove the 5th element from the list.

20) **Object remove(Object obj)**: It removes the specified object from the list.

llistobj.remove("Test Item");

21) **Object removeFirst()**: It removes the first item from the list.

llistobj.removeFirst();

22) **Object removeLast()**: It removes the last item of the list.

llistobj.removeLast();

23) **Object removeFirstOccurrence(Object item)**: It removes the first occurrence of the specified item.

llistobj.removeFirstOccurrence("text");

It will remove the first occurrence of the string “text” from the list.

24) **Object removeLastOccurrence(Object item)**: It removes the last occurrence of the given element.

llistobj.removeLastOccurrence("String1);

It will remove the last occurrence of string “String1”.

25) **Object set(int index, Object item)**: It updates the item of specified index with the give value.

llistobj.set(2, "Test");

It will update the 3rd element with the string “Test”.

26)**int size()**: It returns the number of elements of the list.

llistobj.size();

# Difference between ArrayList and LinkedList

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a **doubly linked list** to store the elements. |
| 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) 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. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

* ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

### When to use LinkedList and when to use ArrayList?

1) As explained above the insert and remove operations give good performance (O(1)) in LinkedList compared to ArrayList(O(n)). Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

2) Search (get method) operations are fast in Arraylist (O(1)) but not in LinkedList (O(n)) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best bet.

## **Vector**

* **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.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection3{

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

Vector<String> v=**new** Vector<String>();

v.add("Ayush");

v.add("Amit");

v.add("Ashish");

v.add("Garima");

Iterator<String> itr=v.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Ayush

Amit

Ashish

Garima

## Commonly used methods of Vector Class:

1. **void addElement(Object element):** It inserts the element at the end of the Vector.
2. **int capacity():** This method returns the current capacity of the vector.
3. **int size():** It returns the current size of the vector.
4. **void setSize(int size):** It changes the existing size with the specified size.
5. **boolean contains(Object element):** This method checks whether the specified element is present in the Vector. If the element is been found it returns true else false.
6. **boolean containsAll(Collection c):** It returns true if all the elements of collection c are present in the Vector.
7. **Object elementAt(int index):** It returns the element present at the specified location in Vector.
8. **Object firstElement():** It is used for getting the first element of the vector.
9. **Object lastElement():** Returns the last element of the array.
10. **Object get(int index):** Returns the element at the specified index.
11. **boolean isEmpty():** This method returns true if Vector doesn’t have any element.
12. **boolean removeElement(Object element):** Removes the specifed element from vector.
13. **boolean removeAll(Collection c):** It Removes all those elements from vector which are present in the Collection c.
14. **void setElementAt(Object element, int index):** It updates the element of specifed index with the given element.

# Difference between ArrayList and Vector In java

BY CHAITANYA SINGH | FILED UNDER: [JAVA COLLECTIONS](https://beginnersbook.com/category/java-collections/)

[ArrayList](https://beginnersbook.com/2013/12/java-arraylist/) and [Vector](https://beginnersbook.com/2013/12/vector-in-java/) both use Array as a data structure internally. However there are few differences in the way they store and process the data. In this post we will discuss the difference and similarities between ArrayList and Vector.

## ArrayList Vs Vector:

1) **Synchronization**: ArrayList is non-synchronized which means multiple threads can work on ArrayList at the same time. For e.g. if one thread is performing an add operation on ArrayList, there can be an another thread performing remove operation on ArrayList at the same time in a multithreaded environment

while Vector is synchronized. This means if one thread is working on Vector, no other thread can get a hold of it. Unlike ArrayList, only one thread can perform an operation on vector at a time.

2) **Resize:** Both ArrayList and Vector can grow and shrink dynamically to maintain the optimal use of storage, however the way they resized is different. ArrayList grow by half of its size when resized while Vector doubles the size of itself by default when grows.

3) **Performance**: ArrayList gives better performance as it is non-synchronized. Vector operations gives poor performance as they are thread-safe, the thread which works on Vector gets a lock on it which makes other thread wait till the lock is released.

4) **fail-fast**: First let me explain what is fail-fast: If the collection (ArrayList, vector etc) gets structurally modified by any means, except the **add or remove methods** of iterator, after creation of iterator then the iterator will throw [ConcurrentModificationException](https://docs.oracle.com/javase/6/docs/api/java/util/ConcurrentModificationException.html). Structural modification refers to the addition or deletion of elements from the collection.

As per the [Vector javadoc](https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html) the Enumeration returned by Vector is not fail-fast. On the other side the iterator and listIterator returned by ArrayList are fail-fast.

5) **Who belongs to collection framework really?**The vector was not the part of collection framework, it has been included in collections later. It can be considered as Legacy code. There is nothing about Vector which List collection cannot do. Therefore Vector should be avoided. If there is a need of thread-safe operation make ArrayList synchronized as discussed in the next section of this post or use [CopyOnWriteArrayList](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html) which is a thread-safe variant of ArrayList.

There are few **similarities between** these classes which are as follows:

1. Both Vector and ArrayList use growable array data structure.
2. The iterator and listIterator returned by these classes (Vector and ArrayList) are fail-fast.
3. They both are ordered collection classes as they maintain the elements insertion order.
4. Vector & ArrayList both allows duplicate and null values.
5. They both grows and shrinks automatically when overflow and deletion happens.

### When to use ArrayList and when to use vector?

It totally depends on the requirement. If there is a need to perform “**thread-safe**” operation the vector is your best bet as it ensures that only one thread access the collection at a time.

**Performance:** Synchronized operations consumes more time compared to non-synchronized ones so if there is no need for thread safe operation, ArrayList is a better choice as performance will be improved because of the concurrent processes.

**How to make ArrayList synchronized?**  
As I stated above ArrayList methods are non-synchronized but still if there is a need you can make them synchronized like this –

## **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.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection4{

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

Stack<String> stack = **new** Stack<String>();

stack.push("Ayush");

stack.push("Garvit");

stack.push("Amit");

stack.push("Ashish");

stack.push("Garima");

stack.pop();

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

**while**(itr.hasNext()){

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

}

}

}

Output:

Ayush

Garvit

Amit

Ashish

## **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.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection5{

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

PriorityQueue<String> queue=**new** PriorityQueue<String>();

queue.add("Amit Sharma");

queue.add("Vijay Raj");

queue.add("JaiShankar");

queue.add("Raj");

System.out.println("head:"+queue.element());

System.out.println("head:"+queue.peek());

System.out.println("iterating the queue elements:");

Iterator itr=queue.iterator();

**while**(itr.hasNext()){

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

}

queue.remove();

queue.poll();

System.out.println("after removing two elements:");

Iterator<String> itr2=queue.iterator();

**while**(itr2.hasNext()){

System.out.println(itr2.next());

}

}

}

Output:

head:Amit Sharma

head:Amit Sharma

iterating the queue elements:

Amit Sharma

Raj

JaiShankar

Vijay Raj

after removing two elements:

Raj

Vijay Raj

## Methods of PriorityQueue Class

**boolean add(E e)**: Adds the element into the PriorityQueue.

**void clear()**: This method removes all the elements from the PriorityQueue.

**boolean contains(Element e)**: This method returns true, if the specified element is present in the Queue.

**boolean Offer(E e)**: Same as add() method.

**E peek()**: Returns the head(the first element) of the Queue.

**E poll()**: Removes the head of the Queue and returns it.

**boolean remove(E e)**: This method removes the specified element from the Queue and returns true if the deletion is successful. If the specified element is not present in the Queue then it returns false.

**int size()**: Returns the size of the Queue.

**object[] toArray()**: Returns array containing all the elements of Queue.

## **Deque Interface**

* 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 class**

* 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**.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection6{

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

//Creating Deque and adding elements

Deque<String> deque = **new** ArrayDeque<String>();

deque.add("Gautam");

deque.add("Karan");

deque.add("Ajay");

//Traversing elements

**for** (String str : deque) {

System.out.println(str);

}

}

}

Output:

Gautam

Karan

Ajay

### When to use ArrayList and when to use ArrayDeque?

ArrayDeque has the ability to add or remove the elements from both ends (**head or tail**), on the other hand removing last element from ArrayList takes O(n) time as it traverses the whole list to reach the end. So if you want to add or remove elements from both ends choose ArrayDeque over ArrayList, however if you only want to perform the opreation on the tail (**at the end**) then you should choose ArrayList.

## Methods of Deque interface

**void addFirst(E e)**: Inserts the specified element at the beginning of the Deque.

**void addLast(E e)**: Inserts the specified element at the end of the Deque.

**boolean contains(Object o)**: Returns true if the specified element is present in the Deque.

**E getFirst()**: It returns the first element of the Deque.

**E getLast()**: It returns the last element of the Deque.

**E peekFirst()**: Returns the first element of Deque, or null if the Deque is empty.

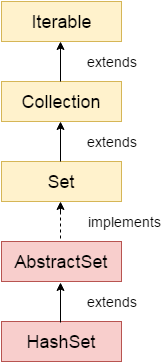
**E peekLast()**: Returns the last element of Deque, or null if the Deque is empty.

**E pollFirst()**: Returns and removes the first element of Deque, or null if the Deque is empty.

**E pollLast()**: Returns and removes the last element of Deque, or null if the Deque is empty.

**int size()**: Returns the number of elements present in Deque.  
[Deque Official Documentation](https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html)

# HashSet class

* used to **create a collection** that uses a **hash table for storage**.
* It **inherits the AbstractSet** class and **implements Set interface**.

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains **unique elements only**.
* HashSet allows **null value**.
* HashSet class is **non synchronized**.
* HashSet **doesn't maintain the insertion order**. Here, elements are inserted on the basis of their **hashcode**.
* HashSet is the best approach for **search operations**.
* The initial **default capacity of HashSet is 16**, and the **load factor is 0.75**.

## **Difference between List and Set**

A list can contain **duplicate** **elements** whereas Set contains **unique elements** only.

### Hierarchy of HashSet class

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection7{

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

//Creating HashSet and adding elements

HashSet<String> set=**new** HashSet<String>();

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

//Traversing elements

Iterator<String> itr=set.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Vijay

Ravi

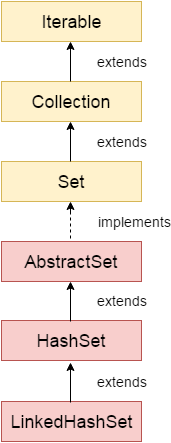
Ajay

### HashSet Methods:

1. **boolean add(Element  e)**: It adds the element e to the list.
2. **void clear()**: It removes all the elements from the list.
3. **Object clone()**: This method returns a shallow copy of the HashSet.
4. **boolean contains(Object o)**: It checks whether the specified Object o is present in the list or not. If the object has been found it returns true else false.
5. **boolean isEmpty()**: Returns true if there is no element present in the Set.
6. **int size()**: It gives the number of elements of a Set.
7. **boolean(Object o)**: It removes the specified Object o from the Set.

# LinkedHashSet class

* Java LinkedHashSet class is a **Hashtable** and **Linked list implementation of the set interface.**
* It **inherits HashSet** class and **implements Set interface**.

The important points about Java LinkedHashSet class are:

* Java LinkedHashSet class contains **unique elements** only like HashSet.
* Java LinkedHashSet class provides **all optional set operation** and **permits null elements.**
* Java LinkedHashSet class is **non synchronized.**
* Java LinkedHashSet class maintains **insertion order**.

## **Hierarchy of LinkedHashSet class**

The LinkedHashSet class extends HashSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection8{

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

LinkedHashSet<String> set=**new** LinkedHashSet<String>();

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

Iterator<String> itr=set.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Ravi

Vijay

Ajay

## **SortedSet Interface**

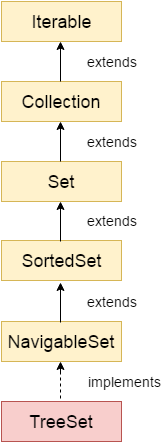
* SortedSet is the alternate of Set interface that provides a **total ordering on its elements.**
* The elements of the SortedSet are arranged in the **increasing (ascending) order.**
* The SortedSet provides the **additional methods** that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

SortedSet<data-type> set = **new** TreeSet();

# 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.

Consider the following example:

**import** java.util.\*;

**public** **class** TestJavaCollection9{

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

//Creating and adding elements

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

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

//traversing elements

Iterator<String> itr=set.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Ajay

Ravi

Vijay

# 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:

ava Map Hierarchy

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.

|  |  |
| --- | --- |
| **Class** | **Description** |
| [HashMap](https://www.javatpoint.com/java-hashmap) | HashMap is the implementation of Map, but it doesn't maintain any order. |
| [LinkedHashMap](https://www.javatpoint.com/java-linkedhashmap) | LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order. |
| [TreeMap](https://www.javatpoint.com/java-treemap) | TreeMap is the implementation of Map and SortedMap. It maintains ascending order. |

### Useful methods of Map interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| V put(Object key, Object value) | It is used to insert an entry in the map. |
| void putAll(Map map) | It is used to insert the specified map in the map. |
| V putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| V remove(Object key) | It is used to delete an entry for the specified key. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| Set keySet() | It returns the Set view containing all the keys. |
| Set<Map.Entry<K,V>> entrySet() | It returns the Set view containing all the keys and values. |
| void clear() | It is used to reset the map. |
| V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| boolean containsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
| boolean containsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| boolean equals(Object o) | It is used to compare the specified Object with the Map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key. |
| int hashCode() | It returns the hash code value for the Map |
| boolean isEmpty() | This method returns true if the map is empty; returns false if it contains at least one key. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection values() | It returns a collection view of the values contained in the map. |
| int size() | This method returns the number of entries in the map. |

## **Map.Entry Interface**

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name. It returns a collection-view of the map, whose elements are of this class. It provides methods to get key and value.

### Methods of Map.Entry interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| K getKey() | It is used to obtain a key. |
| V getValue() | It is used to obtain value. |
| int hashCode() | It is used to obtain hashCode. |
| V setValue(V value) | It is used to replace the value corresponding to this entry with the specified value. |
| boolean equals(Object o) | It is used to compare the specified object with the other existing objects. |
| static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>> comparingByKey() | It returns a comparator that compare the objects in natural order on key. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super K> cmp) | It returns a comparator that compare the objects by key using the given Comparator. |
| static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>> comparingByValue() | It returns a comparator that compare the objects in natural order on value. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super V> cmp) | It returns a comparator that compare the objects by value using the given Comparator. |

### Java Map Example: Generic (New Style)

**import** java.util.\*;

**class** MapExample2{

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

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

  map.put(100,"Amit");

  map.put(101,"Vijay");

  map.put(102,"Rahul");

  //Elements can traverse in any order

**for**(Map.Entry m:map.entrySet()){

   System.out.println(m.getKey()+" "+m.getValue());

  }

 }

}

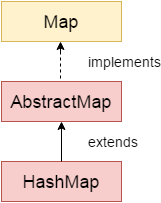
Output:

102 Rahul

100 Amit

101 Vijay

# Java HashMap class



Java HashMap class implements the map interface by using a hash table. It inherits AbstractMap class and implements Map interface.

### Points to remember

* Java HashMap class contains values based on the key.
* Java HashMap class contains only unique keys.
* Java HashMap class may have one null key and multiple null values.
* Java HashMap class is non synchronized.
* Java HashMap class maintains no order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

### Hierarchy of HashMap class

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.

## HashMap Class Methods

Here is the list of methods available in HashMap class. I have also covered examples using these methods at the end of this post.

1. **void clear()**: It removes all the key and value pairs from the specified Map.
2. **Object clone()**: It returns a copy of all the mappings of a map and used for cloning them into another map.
3. **boolean containsKey(Object key)**: It is a boolean function which returns true or false based on whether the specified key is found in the map.
4. **boolean containsValue(Object Value)**: Similar to containsKey() method, however it looks for the specified value instead of key.
5. **Value get(Object key)**: It returns the value for the specified key.
6. **boolean isEmpty()**: It checks whether the map is empty. If there are no key-value mapping present in the map then this function returns true else false.
7. **Set keySet()**: It returns the Set of the keys fetched from the map.
8. **value put(Key k, Value v)**: Inserts key value mapping into the map. Used in the above example.
9. **int size()**: Returns the size of the map – Number of key-value mappings.
10. **Collection values()**: It returns a collection of values of map.
11. **Value remove(Object key)**: It removes the key-value pair for the specified key. Used in the above example.
12. **void putAll(Map m)**: Copies all the elements of a map to the another specified map.

### Java HashMap example to add() elements

Here, we see different ways to insert elements.

**import** java.util.\*;

**class** HashMap1{

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

   HashMap<Integer,String> hm=**new** HashMap<Integer,String>();

    System.out.println("Initial list of elements: "+hm);

      hm.put(100,"Amit");

      hm.put(101,"Vijay");

      hm.put(102,"Rahul");

      System.out.println("After invoking put() method ");

**for**(Map.Entry m:hm.entrySet()){

       System.out.println(m.getKey()+" "+m.getValue());

      }

      hm.putIfAbsent(103, "Gaurav");

      System.out.println("After invoking putIfAbsent() method ");

**for**(Map.Entry m:hm.entrySet()){

           System.out.println(m.getKey()+" "+m.getValue());

          }

      HashMap<Integer,String> map=**new** HashMap<Integer,String>();

      map.put(104,"Ravi");

      map.putAll(hm);

      System.out.println("After invoking putAll() method ");

**for**(Map.Entry m:map.entrySet()){

           System.out.println(m.getKey()+" "+m.getValue());

          }

 }

}

Initial list of elements: {}

After invoking put() method

100 Amit

101 Vijay

102 Rahul

After invoking putIfAbsent() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

After invoking putAll() method

100 Amit

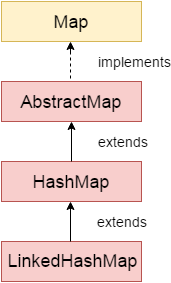
101 Vijay

102 Rahul

103 Gaurav

104 Ravi

# Java LinkedHashMap class



Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

### Points to remember

* Java LinkedHashMap contains values based on the key.
* Java LinkedHashMap contains unique elements.
* Java LinkedHashMap may have one null key and multiple null values.
* Java LinkedHashMap is non synchronized.
* Java LinkedHashMap maintains insertion order.
* The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

### Methods of Java LinkedHashMap class

|  |  |
| --- | --- |
| **Method** | **Description** |
| V get(Object key) | It returns the value to which the specified key is mapped. |
| void clear() | It removes all the key-value pairs from a map. |
| boolean containsValue(Object value) | It returns true if the map maps one or more keys to the specified value. |
| Set<Map.Entry<K,V>> entrySet() | It returns a Set view of the mappings contained in the map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped or defaultValue if this map contains no mapping for the key. |
| Set<K> keySet() | It returns a Set view of the keys contained in the map |
| protected boolean removeEldestEntry(Map.Entry<K,V> eldest) | It returns true on removing its eldest entry. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection<V> values() | It returns a Collection view of the values contained in this map. |

### Java LinkedHashMap Example

**import** java.util.\*;

**class** LinkedHashMap1{

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

  LinkedHashMap<Integer,String> hm=**new** LinkedHashMap<Integer,String>();

  hm.put(100,"Amit");

  hm.put(101,"Vijay");

  hm.put(102,"Rahul");

**for**(Map.Entry m:hm.entrySet()){

   System.out.println(m.getKey()+" "+m.getValue());

  }

 }

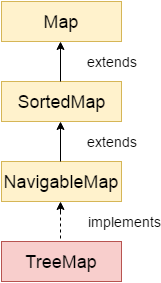
}

Output:100 Amit

101 Vijay

102 Rahul

# Java TreeMap class



Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

The important points about Java TreeMap class are:

* Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* Java TreeMap contains only unique elements.
* Java TreeMap cannot have a null key but can have multiple null values.
* Java TreeMap is non synchronized.
* Java TreeMap maintains ascending order.

### Methods of Java TreeMap class

|  |  |
| --- | --- |
| **Method** | **Description** |
| Map.Entry<K,V> ceilingEntry(K key) | It returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key. |
| K ceilingKey(K key) | It returns the least key, greater than the specified key or null if there is no such key. |
| void clear() | It removes all the key-value pairs from a map. |
| Object clone() | It returns a shallow copy of TreeMap instance. |
| Comparator<? super K> comparator() | It returns the comparator that arranges the key in order, or null if the map uses the natural ordering. |
| NavigableSet<K> descendingKeySet() | It returns a reverse order NavigableSet view of the keys contained in the map. |
| NavigableMap<K,V> descendingMap() | It returns the specified key-value pairs in descending order. |
| Map.Entry firstEntry() | It returns the key-value pair having the least key. |
| Map.Entry<K,V> floorEntry(K key) | It returns the greatest key, less than or equal to the specified key, or null if there is no such key. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| SortedMap<K,V> headMap(K toKey) | It returns the key-value pairs whose keys are strictly less than toKey. |
| NavigableMap<K,V> headMap(K toKey, boolean inclusive) | It returns the key-value pairs whose keys are less than (or equal to if inclusive is true) toKey. |
| Map.Entry<K,V> higherEntry(K key) | It returns the least key strictly greater than the given key, or null if there is no such key. |
| K higherKey(K key) | It is used to return true if this map contains a mapping for the specified key. |
| Set keySet() | It returns the collection of keys exist in the map. |
| Map.Entry<K,V> lastEntry() | It returns the key-value pair having the greatest key, or null if there is no such key. |
| Map.Entry<K,V> lowerEntry(K key) | It returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key. |
| K lowerKey(K key) | It returns the greatest key strictly less than the given key, or null if there is no such key. |
| NavigableSet<K> navigableKeySet() | It returns a NavigableSet view of the keys contained in this map. |
| Map.Entry<K,V> pollFirstEntry() | It removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty. |
| Map.Entry<K,V> pollLastEntry() | It removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty. |
| V put(K key, V value) | It inserts the specified value with the specified key in the map. |
| void putAll(Map<? extends K,? extends V> map) | It is used to copy all the key-value pair from one map to another map. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive) | It returns key-value pairs whose keys range from fromKey to toKey. |
| SortedMap<K,V> subMap(K fromKey, K toKey) | It returns key-value pairs whose keys range from fromKey, inclusive, to toKey, exclusive. |
| SortedMap<K,V> tailMap(K fromKey) | It returns key-value pairs whose keys are greater than or equal to fromKey. |
| NavigableMap<K,V> tailMap(K fromKey, boolean inclusive) | It returns key-value pairs whose keys are greater than (or equal to, if inclusive is true) fromKey. |
| boolean containsKey(Object key) | It returns true if the map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It returns true if the map maps one or more keys to the specified value. |
| K firstKey() | It is used to return the first (lowest) key currently in this sorted map. |
| V get(Object key) | It is used to return the value to which the map maps the specified key. |
| K lastKey() | It is used to return the last (highest) key currently in the sorted map. |
| V remove(Object key) | It removes the key-value pair of the specified key from the map. |
| Set<Map.Entry<K,V>> entrySet() | It returns a set view of the mappings contained in the map. |
| int size() | It returns the number of key-value pairs exists in the hashtable. |
| Collection values() | It returns a collection view of the values contained in the map. |

### Java TreeMap Example

1. **import** java.util.\*;
2. **class** TreeMap1{
3. **public** **static** **void** main(String args[]){
4. TreeMap<Integer,String> map=**new** TreeMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(102,"Ravi");
7. map.put(101,"Vijay");
8. map.put(103,"Rahul");
10. **for**(Map.Entry m:map.entrySet()){
11. System.out.println(m.getKey()+" "+m.getValue());
12. }
13. }
14. }

Output:100 Amit

101 Vijay

102 Ravi

103 Rahul

# Java Hashtable class

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

### Points to remember

* 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.

#### Methods of Hashtable class:

1) void clear(): Removes all the key-value mappings from Hashtable and makes it empty. Clears this hashtable so that it contains no keys..

2) Object clone(): Creates a shallow copy of this hashtable. All the structure of the hashtable itself is copied, but the keys and values are not cloned. This is a relatively expensive operation.

3) boolean contains(Object value): Tests if some key maps into the specified value in this hashtable. This operation is more expensive than the containsKey method.  
Note that this method is identical in functionality to containsValue, (which is part of the Map interface in the collections framework).

4) boolean isEmpty(): Tests if this hashtable maps no keys to values.

5) Enumeration keys(): Returns an enumeration of the keys contained in the hash table.

6) Object put(Object key, Object value): Maps the specified key to the specified value in this hashtable.

7) void rehash(): Increases the size of the hash table and rehashes all of its keys.

8) Object remove(Object key): Removes the key (and its corresponding value) from this hashtable.

9) int size(): Returns the number of key-value mappings present in Hashtable.

10) String toString(): Returns the string equivalent of a hash table.

11) boolean containsKey(Object key): Tests if the specified object is a key in this hashtable.

12) boolean containsValue(Object value): Tests if the specified object is a value in this hashtable. Returns true if some value equal to value exists within the hash table. Returns false if the value isn’t found.

13) Enumeration elements(): Returns an enumeration of the values contained in the hash table.

14) Object get(Object key): Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

# 34 Java Collections Interview Questions

In Java, collection interview questions are most asked by the interviewers. Here is the list of the most asked collections interview questions with answers.

### 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?

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:**

1. **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:**

1. **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:**

1. **public** **interface** Set<E> **extends** Collection<E>

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

**Syntax:**

1. **public** **interface** Queue<E> **extends** Collection<E>

**4. 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:**

1. **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 Map 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) | ArrayList is not efficient for manipulation because too much is required. | LinkedList is efficient for manipulation. |
| 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. |

### 6) What is the difference between Iterator and ListIterator?

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 impended by hash table whereas TreeSet implemented by a Tree structure.
* HashSet performs faster than TreeSet.
* HashSet is backed by HashMap whereas TreeSet is backed by TreeMap.

### 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?

|  |  |  |
| --- | --- | --- |
| **No.** | **HashMap** | **Hashtable** |
| 1) | HashMap is not synchronized. | Hashtable is synchronized. |
| 2) | HashMap can contain one null key and multiple null values. | Hashtable cannot contain any null key or null value. |
| 3) | HashMap is not ?thread-safe,? so it is useful for non-threaded applications. | Hashtable is thread-safe, and it can be shared between various threads. |
| 4) | 4) HashMap inherits the AbstractMap class | Hashtable inherits the 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. |

### 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:**

1. **public** **interface** BlockingQueue<E> **extends** Queue <E>

### 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.

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** Test {
4. **public** **static** **void** main(String[] args)**throws** Exception{
5. FileReader reader=**new** FileReader("db.properties");
7. Properties p=**new** Properties();
8. p.load(reader);
10. System.out.println(p.getProperty("user"));
11. System.out.println(p.getProperty("password"));
12. }
13. }

**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**

1. Int [] array = **new** **int**[4];
2. System.out.println("The size of the array is " + array.length);

**Finding the size of the ArrayList**

1. ArrayList<String> list=**new** ArrayList<String>();
2. list.add("ankit");
3. list.add("nippun");
4. 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:

1. 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.

1. 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.

### 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.

1. **import** java.util.ArrayList;
2. **import** java.util.Collection;
3. **import** java.util.Collections;
4. **import** java.util.Iterator;
5. **import** java.util.List;
6. **public** **class** ReverseArrayList {
7. **public** **static** **void** main(String[] args) {
8. List list = **new** ArrayList<>();
9. list.add(10);
10. list.add(50);
11. list.add(30);
12. Iterator i = list.iterator();
13. System.out.println("printing the list....");
14. **while**(i.hasNext())
15. {
16. System.out.println(i.next());
17. }
18. Iterator i2 = list.iterator();
19. Collections.reverse(list);
20. System.out.println("printing list in reverse order....");
21. **while**(i2.hasNext())
22. {
23. System.out.println(i2.next());
24. }
25. }
26. }

**Output**

printing the list....

10

50

30

printing list in reverse order....

30

50

10

### 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.

1. **import** java.util.ArrayList;
2. **import** java.util.Collection;
3. **import** java.util.Collections;
4. **import** java.util.Comparator;
5. **import** java.util.Iterator;
6. **import** java.util.List;
8. **public** **class** ReverseArrayList {
9. **public** **static** **void** main(String[] args) {
10. List list = **new** ArrayList<>();
11. list.add(10);
12. list.add(50);
13. list.add(30);
14. list.add(60);
15. list.add(20);
16. list.add(90);
18. Iterator i = list.iterator();
19. System.out.println("printing the list....");
20. **while**(i.hasNext())
21. {
22. System.out.println(i.next());
23. }
25. Comparator cmp = Collections.reverseOrder();
26. Collections.sort(list,cmp);
27. System.out.println("printing list in descending order....");
28. Iterator i2 = list.iterator();
29. **while**(i2.hasNext())
30. {
31. System.out.println(i2.next());
32. }
34. }
35. }

**Output**

printing the list....

10

50

30

60

20

90

printing list in descending order....

90

60

50

30

20

10

### 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.