Collections in Java

The **Collection in Java** is 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](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-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](https://www.javatpoint.com/object-and-class-in-java) and [interfaces](https://www.javatpoint.com/interface-in-java) for the Collection framework.



|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the  invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | Public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified  collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection  that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking  collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from  the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |

|  |  |  |
| --- | --- | --- |
| 19 | public int hashCode() | It returns the hash code number of the collection |

### **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**

The Iterable interface is the 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.,

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

## **Collection Interface**

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

## **List Interface**

List interface is the 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 :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. 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.

## **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. Consider the following example.

1. **import** java.util.\*;
2. **class** TestJavaCollection1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Ravi");//Adding object in arraylist
6. list.add("Vijay");
7. list.add("Ravi");
8. list.add("Ajay");
9. //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }

Output:

Ravi

Vijay

Ravi

Ajay

## **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.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection2{
3. **public** **static** **void** main(String args[]){
4. LinkedList<String> al=**new** LinkedList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

Output:

Ravi

Vijay

Ravi

Ajay

## **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.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection3{
3. **public** **static** **void** main(String args[]){
4. Vector<String> v=**new** Vector<String>();
5. v.add("Ayush");
6. v.add("Amit");
7. v.add("Ashish");
8. v.add("Garima");
9. Iterator<String> itr=v.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

Output:

Ayush

Amit

Ashish

Garima

## **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.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection4{
3. **public** **static** **void** main(String args[]){
4. Stack<String> stack = **new** Stack<String>();
5. stack.push("Ayush");
6. stack.push("Garvit");
7. stack.push("Amit");
8. stack.push("Ashish");
9. stack.push("Garima");
10. stack.pop();
11. Iterator<String> itr=stack.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Ayush

Garvit

Amit

Ashish

## **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:

1. Queue<String> q1 = **new** PriorityQueue();
2. Queue<String> q2 = **new** ArrayDeque();

There are various classes that implement the Queue interface, some of them are given below.

## **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.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection5{
3. **public** **static** **void** main(String args[]){
4. PriorityQueue<String> queue=**new** PriorityQueue<String>();
5. queue.add("Amit Sharma");
6. queue.add("Vijay Raj");
7. queue.add("JaiShankar");
8. queue.add("Raj");
9. System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
20. **while**(itr2.hasNext()){
21. System.out.println(itr2.next());
22. }
23. }
24. }

Output:

head:Amit Sharma

head:Amit Sharma

iterating the queue elements:

Amit Sharma

Raj

JaiShankar

Vijay Raj

after removing two elements:

Raj

Vijay Raj

## **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:

1. 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.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection6{
3. **public** **static** **void** main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = **new** ArrayDeque<String>();
6. deque.add("Gautam");
7. deque.add("Karan");
8. deque.add("Ajay");
9. //Traversing elements
10. **for** (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }

Output:

Gautam

Karan

Ajay

## **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.

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

## **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.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection7{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output:

Vijay

Ravi

Ajay

## **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.

Consider the following example.

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection8{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> set=**new** LinkedHashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ravi");
8. set.add("Ajay");
9. Iterator<String> itr=set.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

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:

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

## **TreeSet**

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

1. **import** java.util.\*;
2. **public** **class** TestJavaCollection9{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> set=**new** TreeSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

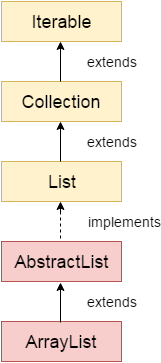
Output:

Ajay

Ravi

Vijay

# Java ArrayList



Java **ArrayList** class uses a dynamic [*array*](https://www.javatpoint.com/array-in-java) for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the java.util package. It is like the Vector in C++.

The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of the List interface here. The ArrayList maintains the insertion order internally.

It inherits the AbstractList class and implements [List interface](https://www.javatpoint.com/java-list).

The important points about the Java ArrayList class are:

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non [synchronized](https://www.javatpoint.com/synchronization-in-java).
* Java ArrayList allows random access because the array works on an index basis.
* In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.
* We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:

1. ArrayList<**int**> al = ArrayList<**int**>(); // does not work
2. ArrayList<Integer> al = **new** ArrayList<Integer>(); // works fine

* Java ArrayList gets initialized by the size. The size is dynamic in the array list, which varies according to the elements getting added or removed from the list.

### **Hierarchy of ArrayList class**

As shown in the above diagram, the Java ArrayList class extends AbstractList class which implements the List interface. The List interface extends the [Collection](https://www.javatpoint.com/collections-in-java) and Iterable interfaces in hierarchical order.

### **ArrayList class declaration**

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

1. **public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable

### **Constructors of ArrayList**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| ArrayList() | It is used to build an empty array list. |
| ArrayList(Collection<? extends E> c) | It is used to build an array list that is initialized with the elements of the  collection c. |
| ArrayList(int capacity) | It is used to build an array list that has the specified initial capacity. |

### **Methods of ArrayList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void [add](https://www.javatpoint.com/java-arraylist-add-method)(int index, E element) | It is used to insert the specified element at the specified position in a list. |
| boolean [add](https://www.javatpoint.com/java-arraylist-add-method)(E e) | It is used to append the specified element at the end of a list. |
| boolean [addAll](https://www.javatpoint.com/java-arraylist-addall-method)(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the  end of this list, in the order that they are returned by the specified  collection's iterator. |
| boolean [addAll](https://www.javatpoint.com/java-arraylist-addall-method)(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection,  starting at the specified position of the list. |
| void [clear](https://www.javatpoint.com/java-arraylist-clear-method)() | It is used to remove all of the elements from this list. |
| void ensureCapacity(int requiredCapacity) | It is used to enhance the capacity of an ArrayList instance. |
| E get(int index) | It is used to fetch the element from the particular position of the list. |
| boolean isEmpty() | It returns true if the list is empty, otherwise false. |
| [Iterator()](https://www.javatpoint.com/java-arraylist-iterator-method) |  |
| [listIterator()](https://www.javatpoint.com/java-arraylist-listiterator-method) |  |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the  specified element, or -1 if the list does not contain this element. |
| Object[] toArray() | It is used to return an array containing all of the elements in this list in the  correct order. |
| <T> T[] toArray(T[] a) | It is used to return an array containing all of the elements in this list in the  correct order. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| boolean contains(Object o) | It returns true if the list contains the specified element. |
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the  specified element, or -1 if the List does not contain this element. |
| E remove(int index) | It is used to remove the element present at the specified position in the list. |
| boolean [remove](https://www.javatpoint.com/java-arraylist-remove-method)(Object o) | It is used to remove the first occurrence of the specified element. |
| boolean [removeAll](https://www.javatpoint.com/java-arraylist-removeall-method)(Collection<?> c) | It is used to remove all the elements from the list. |
| boolean removeIf(Predicate<? super E> filter) | It is used to remove all the elements from the list that satisfies the given  predicate. |
| protected void [removeRange](https://www.javatpoint.com/java-arraylist-removerange-method)(int fromIndex, int toIndex) | It is used to remove all the elements lies within the given range. |
| void replaceAll(UnaryOperator<E> operator) | It is used to replace all the elements from the list with the specified  element. |
| void [retainAll](https://www.javatpoint.com/java-arraylist-retainall-method)(Collection<?> c) | It is used to retain all the elements in the list that are present in the  specified collection. |
| E set(int index, E element) | It is used to replace the specified element in the list, present at the  specified position. |
| void sort(Comparator<? super E> c) | It is used to sort the elements of the list on the basis of the  specified comparator. |
| Spliterator<E> spliterator() | It is used to create a spliterator over the elements in a list. |
| List<E> subList(int fromIndex, int toIndex) | It is used to fetch all the elements that lies within the given range. |
| int size() | It is used to return the number of elements present in the list. |
| void trimToSize() | It is used to trim the capacity of this ArrayList instance to be the  list's current size. |

### **Java Non-generic Vs. Generic Collection**

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in a collection. Now it is type-safe, so typecasting is not required at runtime.

Let's see the old non-generic example of creating a Java collection.

1. ArrayList list=**new** ArrayList();//creating old non-generic arraylist

Let's see the new generic example of creating java collection.

1. ArrayList<String> list=**new** ArrayList<String>();//creating new generic arraylist

In a generic collection, we specify the type in angular braces. Now ArrayList is forced to have the only specified type of object in it. If you try to add another type of object, it gives a compile-time error.

### **Java ArrayList Example**

**FileName:** ArrayListExample1.java

1. **import** java.util.\*;
2. **public** **class** ArrayListExample1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Mango");//Adding object in arraylist
6. list.add("Apple");
7. list.add("Banana");
8. list.add("Grapes");
9. //Printing the arraylist object
10. System.out.println(list);
11. }

**}**

**Output:**

[Mango, Apple, Banana, Grapes]

### **Iterating ArrayList using For-each loop**

Let's see an example to traverse the ArrayList elements using the for-each loop

**FileName:** ArrayListExample3.java

1. **import** java.util.\*;
2. **public** **class** ArrayListExample3{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
5. list.add("Mango");//Adding object in arraylist
6. list.add("Apple");
7. list.add("Banana");
8. list.add("Grapes");
9. //Traversing list through for-each loop
10. **for**(String fruit:list)
11. System.out.println(fruit);
13. }
14. }

**Output:**

Mango

Apple

Banana

Grapes

### **Get and Set ArrayList**

The get() method returns the element at the specified index, whereas the set() method changes the element.

**FileName:** ArrayListExample4.java

1. **import** java.util.\*;
2. **public** **class** ArrayListExample4{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Mango");
6. al.add("Apple");
7. al.add("Banana");
8. al.add("Grapes");
9. //accessing the element
10. System.out.println("Returning element: "+al.get(1));//it will return the 2nd element, because index starts from 0
11. //changing the element
12. al.set(1,"Dates");
13. //Traversing list
14. **for**(String fruit:al)
15. System.out.println(fruit);
17. }
18. }

**Output:**

Returning element: Apple

Mango

Dates

Banana

Grapes

### **How to Sort ArrayList**

The java.util package provides a utility class **Collections**, which has the static method sort(). Using the **Collections.sort()** method, we can easily sort the ArrayList.

**FileName:** SortArrayList.java

1. **import** java.util.\*;
2. **class** SortArrayList{
3. **public** **static** **void** main(String args[]){
4. //Creating a list of fruits
5. List<String> list1=**new** ArrayList<String>();
6. list1.add("Mango");
7. list1.add("Apple");
8. list1.add("Banana");
9. list1.add("Grapes");
10. //Sorting the list
11. Collections.sort(list1);
12. //Traversing list through the for-each loop
13. **for**(String fruit:list1)
14. System.out.println(fruit);
16. System.out.println("Sorting numbers...");
17. //Creating a list of numbers
18. List<Integer> list2=**new** ArrayList<Integer>();
19. list2.add(21);
20. list2.add(11);
21. list2.add(51);
22. list2.add(1);
23. //Sorting the list
24. Collections.sort(list2);
25. //Traversing list through the for-each loop
26. **for**(Integer number:list2)
27. System.out.println(number);
28. }
30. }

**Output:**

Apple

Banana

Grapes

Mango

Sorting numbers...

1

11

21

51

### **Ways to iterate the elements of the collection in Java**

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.
7. **import** java.util.\*;
8. **class** ArrayList4{
9. **public** **static** **void** main(String args[]){
10. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist
11. list.add("Ravi");//Adding object in arraylist
12. list.add("Vijay");
13. list.add("Ravi");
14. list.add("Ajay");
16. System.out.println("Traversing list through List Iterator:");
17. //Here, element iterates in reverse order
18. ListIterator<String> list1=list.listIterator(list.size());
19. **while**(list1.hasPrevious())
20. {
21. String str=list1.previous();
22. System.out.println(str);
23. }
24. System.out.println("Traversing list through for loop:");
25. **for**(**int** i=0;i<list.size();i++)
26. {
27. System.out.println(list.get(i));
28. }
30. System.out.println("Traversing list through forEach() method:");
31. //The forEach() method is a new feature, introduced in Java 8.
32. list.forEach(a->{ //Here, we are using lambda expression
33. System.out.println(a);
34. });
36. System.out.println("Traversing list through forEachRemaining() method:");
37. Iterator<String> itr=list.iterator();
38. itr.forEachRemaining(a-> //Here, we are using lambda expression
39. {
40. System.out.println(a);
41. });
42. }
43. }

**Output:**

Traversing list through List Iterator:

Ajay

Ravi

Vijay

Ravi

Traversing list through for loop:

Ravi

Vijay

Ravi

Ajay

Traversing list through forEach() method:

Ravi

Vijay

Ravi

Ajay

Traversing list through forEachRemaining() method:

Ravi

Vijay

Ravi

Ajay

### **User-defined class objects in Java ArrayList**

Let's see an example where we are storing Student class object in an array list.

**FileName:** ArrayList5.java

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }
11. **import** java.util.\*;
12. **class** ArrayList5{
13. **public** **static** **void** main(String args[]){
14. //Creating user-defined class objects
15. Student s1=**new** Student(101,"Sonoo",23);
16. Student s2=**new** Student(102,"Ravi",21);
17. Student s2=**new** Student(103,"Hanumat",25);
18. //creating arraylist
19. ArrayList<Student> al=**new** ArrayList<Student>();
20. al.add(s1);//adding Student class object
21. al.add(s2);
22. al.add(s3);
23. //Getting Iterator
24. Iterator itr=al.iterator();
25. //traversing elements of ArrayList object
26. **while**(itr.hasNext()){
27. Student st=(Student)itr.next();
28. System.out.println(st.rollno+" "+st.name+" "+st.age);
29. }
30. }
31. }

**Output:**

101 Sonoo 23

102 Ravi 21

103 Hanumat 25

### **Java ArrayList Serialization and Deserialization Example**

Let's see an example to serialize an ArrayList object and then deserialize it.

**FileName:** ArrayList6.java

1. **import** java.io.\*;
2. **import** java.util.\*;
3. **class** ArrayList6 {
5. **public** **static** **void** main(String [] args)
6. {
7. ArrayList<String> al=**new** ArrayList<String>();
8. al.add("Ravi");
9. al.add("Vijay");
10. al.add("Ajay");
12. **try**
13. {
14. //Serialization
15. FileOutputStream fos=**new** FileOutputStream("file");
16. ObjectOutputStream oos=**new** ObjectOutputStream(fos);
17. oos.writeObject(al);
18. fos.close();
19. oos.close();
20. //Deserialization
21. FileInputStream fis=**new** FileInputStream("file");
22. ObjectInputStream ois=**new** ObjectInputStream(fis);
23. ArrayList  list=(ArrayList)ois.readObject();
24. System.out.println(list);
25. }**catch**(Exception e)
26. {
27. System.out.println(e);
28. }
29. }
30. }

**Output:**

[Ravi, Vijay, Ajay]

### **Java ArrayList example to add elements**

Here, we see different ways to add an element.

**FileName:** ArrayList7.java

1. **import** java.util.\*;
2. **class** ArrayList7{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. System.out.println("Initial list of elements: "+al);
6. //Adding elements to the end of the list
7. al.add("Ravi");
8. al.add("Vijay");
9. al.add("Ajay");
10. System.out.println("After invoking add(E e) method: "+al);
11. //Adding an element at the specific position
12. al.add(1, "Gaurav");
13. System.out.println("After invoking add(int index, E element) method: "+al);
14. ArrayList<String> al2=**new** ArrayList<String>();
15. al2.add("Sonoo");
16. al2.add("Hanumat");
17. //Adding second list elements to the first list
18. al.addAll(al2);
19. System.out.println("After invoking addAll(Collection<? extends E> c) method: "+al);
20. ArrayList<String> al3=**new** ArrayList<String>();
21. al3.add("John");
22. al3.add("Rahul");
23. //Adding second list elements to the first list at specific position
24. al.addAll(1, al3);
25. System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+al);
27. }
28. }

**Output:**

Initial list of elements: []

After invoking add(E e) method: [Ravi, Vijay, Ajay]

After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]

After invoking addAll(Collection<? extends E> c) method:

[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addAll(int index, Collection<? extends E> c) method:

[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

### **Java ArrayList example to remove elements**

Here, we see different ways to remove an element.

**FileName:** ArrayList8.java

1. **import** java.util.\*;
2. **class** ArrayList8 {
4. **public** **static** **void** main(String [] args)
5. {
6. ArrayList<String> al=**new** ArrayList<String>();
7. al.add("Ravi");
8. al.add("Vijay");
9. al.add("Ajay");
10. al.add("Anuj");
11. al.add("Gaurav");
12. System.out.println("An initial list of elements: "+al);
13. //Removing specific element from arraylist
14. al.remove("Vijay");
15. System.out.println("After invoking remove(object) method: "+al);
16. //Removing element on the basis of specific position
17. al.remove(0);
18. System.out.println("After invoking remove(index) method: "+al);
20. //Creating another arraylist
21. ArrayList<String> al2=**new** ArrayList<String>();
22. al2.add("Ravi");
23. al2.add("Hanumat");
24. //Adding new elements to arraylist
25. al.addAll(al2);
26. System.out.println("Updated list : "+al);
27. //Removing all the new elements from arraylist
28. al.removeAll(al2);
29. System.out.println("After invoking removeAll() method: "+al);
30. //Removing elements on the basis of specified condition
31. al.removeIf(contains("Ajay"));   //Here, we are using Lambda expression
32. System.out.println("After invoking removeIf() method: "+al);
33. //Removing all the elements available in the list
34. al.clear();
35. System.out.println("After invoking clear() method: "+al);
36. }
37. }

**Output:**

An initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav]

After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav]

After invoking remove(index) method: [Ajay, Anuj, Gaurav]

Updated list : [Ajay, Anuj, Gaurav, Ravi, Hanumat]

After invoking removeAll() method: [Ajay, Anuj, Gaurav]

After invoking removeIf() method: [Anuj, Gaurav]

After invoking clear() method: []

### **Java ArrayList example of retainAll() method**

**FileName:** ArrayList9.java

1. **import** java.util.\*;
2. **class** ArrayList9{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ajay");
8. ArrayList<String> al2=**new** ArrayList<String>();
9. al2.add("Ravi");
10. al2.add("Hanumat");
11. al.retainAll(al2);
12. System.out.println("iterating the elements after retaining the elements of al2");
13. Iterator itr=al.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
17. }
18. }

**Output:**

iterating the elements after retaining the elements of al2

Ravi

### **Java ArrayList example of isEmpty() method**

**FileName:** ArrayList4.java

1. **import** java.util.\*;
2. **class** ArrayList10{
4. **public** **static** **void** main(String [] args)
5. {
6. ArrayList<String> al=**new** ArrayList<String>();
7. System.out.println("Is ArrayList Empty: "+al.isEmpty());
8. al.add("Ravi");
9. al.add("Vijay");
10. al.add("Ajay");
11. System.out.println("After Insertion");
12. System.out.println("Is ArrayList Empty: "+al.isEmpty());
13. }
14. }

**Output:**

Is ArrayList Empty: true

After Insertion

Is ArrayList Empty: false

### **Java ArrayList Example: Book**

Let's see an ArrayList example where we are adding books to the list and printing all the books.

**FileName:** ArrayListExample20.java

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ArrayListExample20 {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** ArrayList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications and Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

**Output:**

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications and Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

### **Size and Capacity of an ArrayList**

Size and capacity of an array list are the two terms that beginners find confusing. Let's understand it in this section with the help of some examples. Consider the following code snippet.

**FileName:** SizeCapacity.java

1. **import** java.util.\*;
3. **public** **class** SizeCapacity
4. {
6. **public** **static** **void** main(String[] args) **throws** Exception
7. {
9. ArrayList<Integer> al = **new** ArrayList<Integer>();
11. System.out.println("The size of the array is: " + al.size());
12. }
13. }

**Output:**

The size of the array is: 0

# Difference between Array and ArrayList

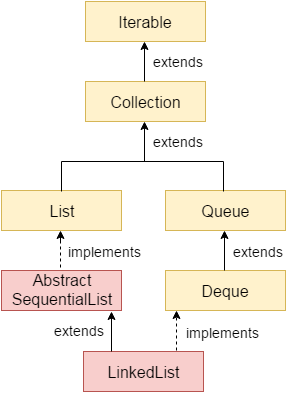
# The above statement creates an array of the specified size. When we try to add more than its size, it throws **ArrayIndexOutOfBoundsException**.

## **Similarities**

* Array and ArrayList both are used for storing elements.
* Array and ArrayList both can store null values.
* They can have duplicate values.
* They do not preserve the order of elements.

|  |  |  |
| --- | --- | --- |
| **Basis** | **Array** | **ArrayList** |
| **Definition** | An **array** is a dynamically-created object.  It serves as a container that holds the constant  number of values of the same type.  It has a contiguous memory location. | The **ArrayList** is a class of Java **Collections**  framework. It contains popular classes like  **Vector, HashTable**, and **HashMap**. |
| **Static/ Dynamic** | Array is **static** in size. | ArrayList is **dynamic** in size. |
| **Resizable** | An array is a **fixed-length** data structure. | ArrayList is a **variable-length** data structure. It can be resized itself when needed. |
| **Initialization** | It is mandatory to provide the size of an  array while initializing it directly or indirectly. | We can create an instance of ArrayList  without specifying its size. Java creates  ArrayList of default size. |
| **Performance** | It performs **fast** in comparison to  ArrayList because of fixed size. | ArrayList is internally backed by the array in  Java. The resize operation in ArrayList slows  down the performance. |
| **Primitive/ Generic type** | An array can store both **objects** and  **primitives** type. | We cannot store **primitive** type in ArrayList.  It automatically converts primitive type to  object. |
| **Iterating Values** | We use **for** loop or **for each** loop to  iterate over an array. | We use an **iterator** to iterate over ArrayList. |
| **Type-Safety** | We cannot use generics along with array  because it is not a convertible type of array. | ArrayList allows us to store only  **generic/ type, that's why it is type-safe.** |
| **Length** | Array provides a **length** variable which  denotes the length of an array. | ArrayList provides the **size()** method to  determine the size of ArrayList. |
| **Adding Elements** | We can add elements in an array by  using the **assignment**operator. | Java provides the **add()** method to add  elements in the ArrayList. |
| **Single/ Multi-Dimensional** | Array can be **multi-dimensional**. | ArrayList is always **single-dimensional**. |

# Java LinkedList class



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.



### **LinkedList class declaration**

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

1. **public** **class** LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>, Cloneable, Serializable

### **Constructors of Java LinkedList**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| LinkedList() | It is used to construct an empty list. |
| LinkedList(Collection<? extends E> c) | It is used to construct a list containing the elements of the  specified collection, in the order, they are  returned by the collection's iterator. |

### **Methods of Java LinkedList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(E e) | It is used to append the specified element to the end of a list. |
| void add(int index, E element) | It is used to insert the specified element at the specified position index in a list. |
| boolean addAll(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the end of this  list, in the order that they are returned by the specified collection's iterator. |
| boolean addAll(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the end of this  list, in the order that they are returned by the specified collection's iterator. |
| boolean addAll(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the  specified position of the list. |
| void addFirst(E e) | It is used to insert the given element at the beginning of a list. |
| void addLast(E e) | It is used to append the given element to the end of a list. |
| void clear() | It is used to remove all the elements from a list. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| boolean contains(Object o) | It is used to return true if a list contains a specified element. |
| Iterator<E> descendingIterator() | It is used to return an iterator over the elements in a deque in reverse sequential order. |
| E element() | It is used to retrieve the first element of a list. |
| E get(int index) | It is used to return the element at the specified position in a list. |
| E getFirst() | It is used to return the first element in a list. |
| E getLast() | It is used to return the last element in a list. |
| int indexOf(Object o) | It is used to return the index in a list of the first occurrence of the specified element,  or -1 if the list does not contain any element. |
| int lastIndexOf(Object o) | It is used to return the index in a list of the last occurrence of the specified element,  or -1 if the list does not contain any element. |
| ListIterator<E> listIterator(int index) | It is used to return a list-iterator of the elements in proper sequence, starting at the  specified position in the list. |
| boolean offer(E e) | It adds the specified element as the last element of a list. |
| boolean offerFirst(E e) | It inserts the specified element at the front of a list. |
| boolean offerLast(E e) | It inserts the specified element at the end of a list. |
| E peek() | It retrieves the first element of a list |
| E peekFirst() | It retrieves the first element of a list or returns null if a list is empty. |
| E peekLast() | It retrieves the last element of a list or returns null if a list is empty. |
| E poll() | It retrieves and removes the first element of a list. |
| E pollFirst() | It retrieves and removes the first element of a list, or returns null if a list is empty. |
| E pollLast() | It retrieves and removes the last element of a list, or returns null if a list is empty. |
| E pop() | It pops an element from the stack represented by a list. |
| void push(E e) | It pushes an element onto the stack represented by a list. |
| E remove() | It is used to retrieve and removes the first element of a list. |
| E remove(int index) | It is used to remove the element at the specified position in a list. |
| boolean remove(Object o) | It is used to remove the first occurrence of the specified element in a list. |
| E removeFirst() | It removes and returns the first element from a list. |
| boolean removeFirstOccurrence(Object o) | It is used to remove the first occurrence of the specified element in a list (when traversing the list from head to tail). |
| E removeLast() | It removes and returns the last element from a list. |
| boolean removeLastOccurrence(Object o) | It removes the last occurrence of the specified element in a list (when traversing the  list from head to tail). |
| E set(int index, E element) | It replaces the element at the specified position in a list with the specified element. |
| Object[] toArray() | It is used to return an array containing all the elements in a list in proper sequence  (from first to the last element). |
| <T> T[] toArray(T[] a) | It returns an array containing all the elements in the proper sequence (from first to  the last element); the runtime type of the returned array is that of the specified array. |
| int size() | It is used to return the number of elements in a list. |

### **Java LinkedList Example**

1. **import** java.util.\*;
2. **public** **class** LinkedList1{
3. **public** **static** **void** main(String args[]){
5. LinkedList<String> al=**new** LinkedList<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Output: Ravi

Vijay

Ravi

Ajay

### **Java LinkedList example to add elements**

Here, we see different ways to add elements.

1. **import** java.util.\*;
2. **public** **class** LinkedList2{
3. **public** **static** **void** main(String args[]){
4. LinkedList<String> ll=**new** LinkedList<String>();
5. System.out.println("Initial list of elements: "+ll);
6. ll.add("Ravi");
7. ll.add("Vijay");
8. ll.add("Ajay");
9. System.out.println("After invoking add(E e) method: "+ll);
10. //Adding an element at the specific position
11. ll.add(1, "Gaurav");
12. System.out.println("After invoking add(int index, E element) method: "+ll);
13. LinkedList<String> ll2=**new** LinkedList<String>();
14. ll2.add("Sonoo");
15. ll2.add("Hanumat");
16. //Adding second list elements to the first list
17. ll.addAll(ll2);
18. System.out.println("After invoking addAll(Collection<? extends E> c) method: "+ll);
19. LinkedList<String> ll3=**new** LinkedList<String>();
20. ll3.add("John");
21. ll3.add("Rahul");
22. //Adding second list elements to the first list at specific position
23. ll.addAll(1, ll3);
24. System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+ll);
25. //Adding an element at the first position
26. ll.addFirst("Lokesh");
27. System.out.println("After invoking addFirst(E e) method: "+ll);
28. //Adding an element at the last position
29. ll.addLast("Harsh");
30. System.out.println("After invoking addLast(E e) method: "+ll);
32. }
33. }

Initial list of elements: []

After invoking add(E e) method: [Ravi, Vijay, Ajay]

After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]

After invoking addAll(Collection<? extends E> c) method:

[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addAll(int index, Collection<? extends E> c) method:

[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addFirst(E e) method:

[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]

After invoking addLast(E e) method:

[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat, Harsh]

### **Java LinkedList example to remove elements**

Here, we see different ways to remove an element.

1. **import** java.util.\*;
2. **public** **class** LinkedList3 {
4. **public** **static** **void** main(String [] args)
5. {
6. LinkedList<String> ll=**new** LinkedList<String>();
7. ll.add("Ravi");
8. ll.add("Vijay");
9. ll.add("Ajay");
10. ll.add("Anuj");
11. ll.add("Gaurav");
12. ll.add("Harsh");
13. ll.add("Virat");
14. ll.add("Gaurav");
15. ll.add("Harsh");
16. ll.add("Amit");
17. System.out.println("Initial list of elements: "+ll);
18. //Removing specific element from arraylist
19. ll.remove("Vijay");
20. System.out.println("After invoking remove(object) method: "+ll);
21. //Removing element on the basis of specific position
22. ll.remove(0);
23. System.out.println("After invoking remove(index) method: "+ll);
24. LinkedList<String> ll2=**new** LinkedList<String>();
25. ll2.add("Ravi");
26. ll2.add("Hanumat");
27. // Adding new elements to arraylist
28. ll.addAll(ll2);
29. System.out.println("Updated list : "+ll);
30. //Removing all the new elements from arraylist
31. ll.removeAll(ll2);
32. System.out.println("After invoking removeAll() method: "+ll);
33. //Removing first element from the list
34. ll.removeFirst();
35. System.out.println("After invoking removeFirst() method: "+ll);
36. //Removing first element from the list
37. ll.removeLast();
38. System.out.println("After invoking removeLast() method: "+ll);
39. //Removing first occurrence of element from the list
40. ll.removeFirstOccurrence("Gaurav");
41. System.out.println("After invoking removeFirstOccurrence() method: "+ll);
42. //Removing last occurrence of element from the list
43. ll.removeLastOccurrence("Harsh");
44. System.out.println("After invoking removeLastOccurrence() method: "+ll);
46. //Removing all the elements available in the list
47. ll.clear();
48. System.out.println("After invoking clear() method: "+ll);
49. }
50. }

Initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking remove(index) method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

Updated list : [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit, Ravi, Hanumat]

After invoking removeAll() method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking removeFirst() method: [Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]

After invoking removeLast() method: [Gaurav, Harsh, Virat, Gaurav, Harsh]

After invoking removeFirstOccurrence() method: [Harsh, Virat, Gaurav, Harsh]

After invoking removeLastOccurrence() method: [Harsh, Virat, Gaurav]

After invoking clear() method: []

### **Java LinkedList Example to reverse a list of elements**

1. **import** java.util.\*;
2. **public** **class** LinkedList4{
3. **public** **static** **void** main(String args[]){
5. LinkedList<String> ll=**new** LinkedList<String>();
6. ll.add("Ravi");
7. ll.add("Vijay");
8. ll.add("Ajay");
9. //Traversing the list of elements in reverse order
10. Iterator i=ll.descendingIterator();
11. **while**(i.hasNext())
12. {
13. System.out.println(i.next());
14. }
16. }
17. }

Output: Ajay

Vijay

Ravi

### **Java LinkedList Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** LinkedListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** LinkedList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



# Difference Between ArrayList and LinkedList

ArrayList and LinkedList both implement the List interface and maintain insertion order. Both are non-synchronized classes.

However, there are many differences between the ArrayList and LinkedList classes that are given below.

|  |  |
| --- | --- |
| **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 other elements 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. |
| 5) The memory location for the elements of an ArrayList is contiguous. | The location for the elements of a linked list is not  contagious. |
| 6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList. | There is no case of default capacity in a LinkedList. In  LinkedList, an empty list is created when a LinkedList is  initialized. |
| 7) To be precise, an ArrayList is a resizable array. | LinkedList implements the doubly linked list of the list  interface. |

# Java Vector

**Vector** is like the dynamic array which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the List interface, so we can use all the methods of List interface here.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

The Iterators returned by the Vector class are fail-fast. In case of concurrent modification, it fails and throws the ConcurrentModificationException.

It is similar to the ArrayList, but with two differences-

* Vector is synchronized.
* Java Vector contains many legacy methods that are not the part of a collections framework.

## **Java Vector class Declaration**

1. **public** **class** Vector<E>
2. **extends** Object<E>
3. **implements** List<E>, Cloneable, Serializable

## **Java Vector Constructors**

Vector class supports four types of constructors. These are given below:

|  |  |  |
| --- | --- | --- |
| **SN** | **Constructor** | **Description** |
| 1) | vector() | It constructs an empty vector with the default size as 10. |
| 2) | vector(int initialCapacity) | It constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero. |
| 3) | vector(int initialCapacity, int capacityIncrement) | It constructs an empty vector with the specified initial capacity and capacity increment. |
| 4) | Vector( Collection<? extends E> c) | It constructs a vector that contains the elements of a collection c. |

### **Java Vector Methods**

The following are the list of Vector class methods:

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1) | [add()](https://www.javatpoint.com/java-vector-add-method) | It is used to append the specified element in the given vector. |
| 2) | [addAll()](https://www.javatpoint.com/java-vector-addall-method) | It is used to append all of the elements in the specified collection to the  end of this Vector. |
| 3) | [addElement()](https://www.javatpoint.com/java-vector-addelement-method) | It is used to append the specified component to the end of this vector.  It increases the vector size by one. |
| 4) | [capacity()](https://www.javatpoint.com/java-vector-capacity-method) | It is used to get the current capacity of this vector. |
| 5) | [clear()](https://www.javatpoint.com/java-vector-clear-method) | It is used to delete all of the elements from this vector. |
| 6) | [clone()](https://www.javatpoint.com/java-vector-clone-method) | It returns a clone of this vector. |
| 7) | [contains()](https://www.javatpoint.com/java-vector-contains-method) | It returns true if the vector contains the specified element. |
| 8) | [containsAll()](https://www.javatpoint.com/java-vector-containsall-method) | It returns true if the vector contains all of the elements in the specified  collection. |
| 9) | [copyInto()](https://www.javatpoint.com/java-vector-copyinto-method) | It is used to copy the components of the vector into the specified array. |
| 10) | [elementAt()](https://www.javatpoint.com/java-vector-elementat-method) | It is used to get the component at the specified index. |
| 11) | [elements()](https://www.javatpoint.com/java-vector-elements-method) | It returns an enumeration of the components of a vector. |
| 12) | [ensureCapacity()](https://www.javatpoint.com/java-vector-ensurecapacity-method) | It is used to increase the capacity of the vector which is in use, if necessary. It ensures that the vector can hold at least the number of components specified by the minimum capacity argument. |
| 13) | [equals()](https://www.javatpoint.com/java-vector-equals-method) | It is used to compare the specified object with the vector for equality. |
| 14) | [firstElement()](https://www.javatpoint.com/java-vector-firstelement-method) | It is used to get the first component of the vector. |
| 15) | [forEach()](https://www.javatpoint.com/java-vector-foreach-method) | It is used to perform the given action for each element of the Iterable until  all elements have been processed or the action throws an exception. |
| 16) | [get()](https://www.javatpoint.com/java-vector-get-method) | It is used to get an element at the specified position in the vector. |
| 17) | [hashCode()](https://www.javatpoint.com/java-vector-hashcode-method) | It is used to get the hash code value of a vector. |
| 18) | [indexOf()](https://www.javatpoint.com/java-vector-indexof-method) | It is used to get the index of the first occurrence of the specified element  in the vector. It returns -1 if the vector does not contain the element. |
| 19) | [insertElementAt()](https://www.javatpoint.com/java-vector-insertelementat-method) | It is used to insert the specified object as a component in the given vector  at the specified index. |
| 20) | [isEmpty()](https://www.javatpoint.com/java-vector-isempty-method) | It is used to check if this vector has no components. |
| 21) | [iterator()](https://www.javatpoint.com/java-vector-iterator-method) | It is used to get an iterator over the elements in the list in proper sequence. |
| 22) | [lastElement()](https://www.javatpoint.com/java-vector-lastelement-method) | It is used to get the last component of the vector. |
| 23) | [lastIndexOf()](https://www.javatpoint.com/java-vector-lastindexof-method) | It is used to get the index of the last occurrence of the specified element  in the vector. It returns -1 if the vector does not contain the element. |
| 24) | listIterator() | It is used to get a list iterator over the elements in the list in proper sequence. |
| 25) | [remove()](https://www.javatpoint.com/java-vector-remove-method) | It is used to remove the specified element from the vector. If the  vector does not contain the element, it is unchanged. |
| 26) | [removeAll()](https://www.javatpoint.com/java-vector-removeall-method) | It is used to delete all the elements from the vector that are present in the  specified collection. |
| 27) | [removeAllElements()](https://www.javatpoint.com/java-vector-removeallelements-method) | It is used to remove all elements from the vector and set the size of the  vector to zero. |
| 28) | [removeElement()](https://www.javatpoint.com/java-vector-removeelement-method) | It is used to remove the first (lowest-indexed) occurrence of the  argument from the vector. |
| 29) | [removeElementAt()](https://www.javatpoint.com/java-vector-removeelementat-method) | It is used to delete the component at the specified index. |
| 30) | removeIf() | It is used to remove all of the elements of the collection that satisfy  the given predicate. |
| 31) | removeRange() | It is used to delete all of the elements from the vector whose index is  between fromIndex, inclusive and toIndex, exclusive. |
| 32) | [replaceAll()](https://www.javatpoint.com/java-vector-replaceall-method) | It is used to replace each element of the list with the result of  applying the operator to that element. |
| 33) | [retainAll()](https://www.javatpoint.com/java-vector-retainall-method) | It is used to retain only that element in the vector which is contained  in the specified collection. |
| 34) | set() | It is used to replace the element at the specified position in the vector  with the specified element. |
| 35) | setElementAt() | It is used to set the component at the specified index of the vector to the  specified object. |
| 36) | setSize() | It is used to set the size of the given vector. |
| 37) | size() | It is used to get the number of components in the given vector. |
| 38) | sort() | It is used to sort the list according to the order induced by the specified  Comparator. |
| 39) | spliterator() | It is used to create a late-binding and fail-fast Spliterator over the elements  in the list. |
| 40) | subList() | It is used to get a view of the portion of the list between fromIndex,  inclusive, and toIndex, exclusive. |
| 41) | toArray() | It is used to get an array containing all of the elements in this vector in  correct order. |
| 42) | toString() | It is used to get a string representation of the vector. |
| 43) | trimToSize() | It is used to trim the capacity of the vector to the vector's current size. |

## **Java Vector Example**

1. **import** java.util.\*;
2. **public** **class** VectorExample {
3. **public** **static** **void** main(String args[]) {
4. //Create a vector
5. Vector<String> vec = **new** Vector<String>();
6. //Adding elements using add() method of List
7. vec.add("Tiger");
8. vec.add("Lion");
9. vec.add("Dog");
10. vec.add("Elephant");
11. //Adding elements using addElement() method of Vector
12. vec.addElement("Rat");
13. vec.addElement("Cat");
14. vec.addElement("Deer");
16. System.out.println("Elements are: "+vec);
17. }
18. }

**Output:**

Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]

## **Java Vector Example 2**

1. **import** java.util.\*;
2. **public** **class** VectorExample1 {
3. **public** **static** **void** main(String args[]) {
4. //Create an empty vector with initial capacity 4
5. Vector<String> vec = **new** Vector<String>(4);
6. //Adding elements to a vector
7. vec.add("Tiger");
8. vec.add("Lion");
9. vec.add("Dog");
10. vec.add("Elephant");
11. //Check size and capacity
12. System.out.println("Size is: "+vec.size());
13. System.out.println("Default capacity is: "+vec.capacity());
14. //Display Vector elements
15. System.out.println("Vector element is: "+vec);
16. vec.addElement("Rat");
17. vec.addElement("Cat");
18. vec.addElement("Deer");
19. //Again check size and capacity after two insertions
20. System.out.println("Size after addition: "+vec.size());
21. System.out.println("Capacity after addition is: "+vec.capacity());
22. //Display Vector elements again
23. System.out.println("Elements are: "+vec);
24. //Checking if Tiger is present or not in this vector
25. **if**(vec.contains("Tiger"))
26. {
27. System.out.println("Tiger is present at the index " +vec.indexOf("Tiger"));
28. }
29. **else**
30. {
31. System.out.println("Tiger is not present in the list.");
32. }
33. //Get the first element
34. System.out.println("The first animal of the vector is = "+vec.firstElement());
35. //Get the last element
36. System.out.println("The last animal of the vector is = "+vec.lastElement());
37. }
38. }

**Output:**

Size is: 4

Default capacity is: 4

Vector element is: [Tiger, Lion, Dog, Elephant]

Size after addition: 7

Capacity after addition is: 8

Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]

Tiger is present at the index 0

The first animal of the vector is = Tiger

The last animal of the vector is = Deer

## **Java Vector Example 3**

1. **import** java.util.\*;
2. **public** **class** VectorExample2 {
3. **public** **static** **void** main(String args[]) {
4. //Create an empty Vector
5. Vector<Integer> in = **new** Vector<>();
6. //Add elements in the vector
7. in.add(100);
8. in.add(200);
9. in.add(300);
10. in.add(200);
11. in.add(400);
12. in.add(500);
13. in.add(600);
14. in.add(700);
15. //Display the vector elements
16. System.out.println("Values in vector: " +in);
17. //use remove() method to delete the first occurence of an element
18. System.out.println("Remove first occourence of element 200: "+in.remove((Integer)200));
19. //Display the vector elements afre remove() method
20. System.out.println("Values in vector: " +in);
21. //Remove the element at index 4
22. System.out.println("Remove element at index 4: " +in.remove(4));
23. System.out.println("New Value list in vector: " +in);
24. //Remove an element
25. in.removeElementAt(5);
26. //Checking vector and displays the element
27. System.out.println("Vector element after removal: " +in);
28. //Get the hashcode for this vector
29. System.out.println("Hash code of this vector = "+in.hashCode());
30. //Get the element at specified index
31. System.out.println("Element at index 1 is = "+in.get(1));
32. }
33. }

**Output:**

Values in vector: [100, 200, 300, 200, 400, 500, 600, 700]

Remove first occourence of element 200: true

Values in vector: [100, 300, 200, 400, 500, 600, 700]

Remove element at index 4: 500

New Value list in vector: [100, 300, 200, 400, 600, 700]

Vector element after removal: [100, 300, 200, 400, 600]

Hash code of this vector = 130123751

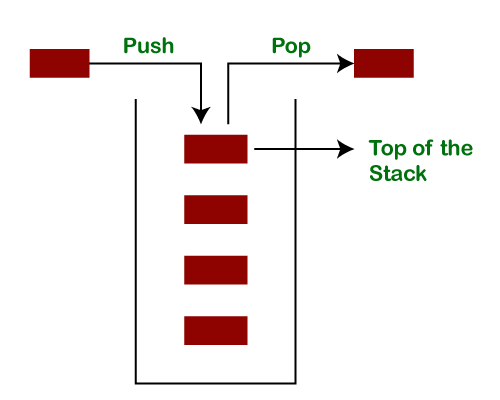
Element at index 1 is = 300

# Java Stack

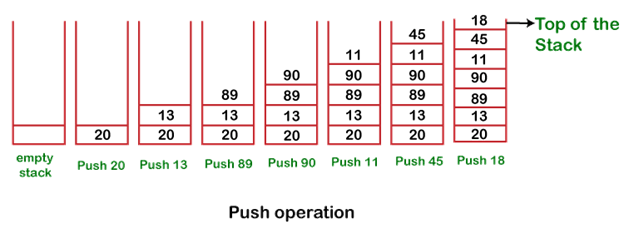
The **stack** is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out** (LIFO). [Java collection](https://www.javatpoint.com/collections-in-java) framework provides many interfaces and classes to store the collection of objects. One of them is the **Stack class** that provides different operations such as push, pop, search, etc.

In this section, we will discuss the **Java Stack class**, its **methods,** and **implement** the stack data structure in a [Java program](https://www.javatpoint.com/java-programs). But before moving to the Java Stack class have a quick view of how the stack works.

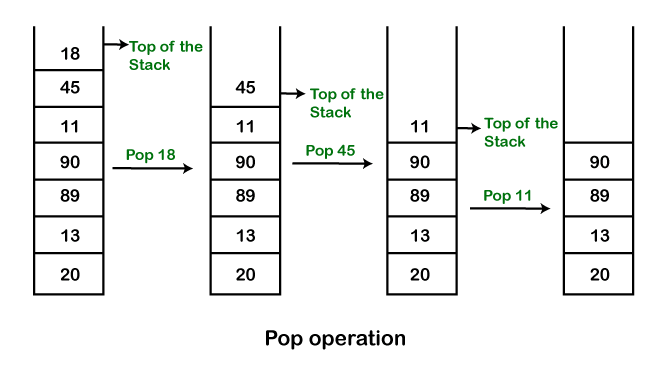
The stack data structure has the two most important operations that are **push** and **pop**. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack. Let's see how they work on stack.



Let's push 20, 13, 89, 90, 11, 45, 18, respectively into the stack.



Let's remove (pop) 18, 45, and 11 from the stack.

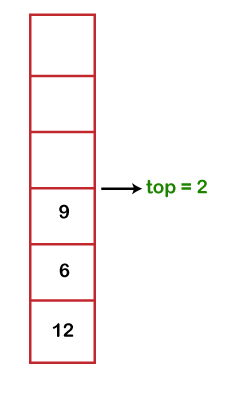


**Empty Stack:** If the stack has no element is known as an **empty stack**. When the stack is empty the value of the top variable is -1.

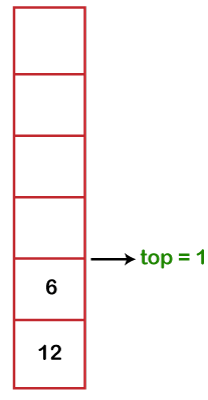


When we push an element into the stack the top is **increased by 1**. In the following figure,

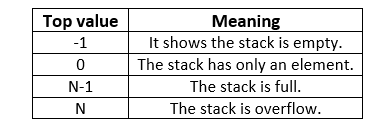
* Push 12, top=0
* Push 6, top=1
* Push 9, top=2



When we pop an element from the stack the value of top is **decreased by 1**. In the following figure, we have popped 9.

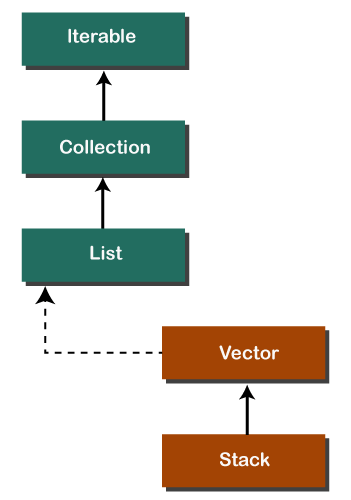


The following table shows the different values of the top.



## **Java Stack Class**

In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class. It also implements interfaces **List, Collection, Iterable, Cloneable, Serializable.** It represents the LIFO stack of objects. Before using the Stack class, we must import the java.util package. The stack class arranged in the Collections framework hierarchy, as shown below.



## **Stack Class Constructor**

The Stack class contains only the **default constructor** that creates an empty stack.

1. **public** Stack()

## **Creating a Stack**

If we want to create a stack, first, import the java.util package and create an object of the Stack class.

1. Stack stk = **new** Stack();

Or

1. Stack<type> stk = **new** Stack<>();

Where type denotes the type of stack like Integer, String, etc.

## **Methods of the Stack Class**

We can perform push, pop, peek and search operation on the stack. The Java Stack class provides mainly five methods to perform these operations. Along with this, it also provides all the methods of the [Java Vector class](https://www.javatpoint.com/java-vector).

|  |  |  |
| --- | --- | --- |
| **Method** | **Modifier and Type** | **Method Description** |
| [empty()](https://www.javatpoint.com/java-stack#empty) | boolean | The method checks the stack is empty or not. |
| [push(E item)](https://www.javatpoint.com/java-stack#push) | E | The method pushes (insert) an element onto the top of the stack. |
| [pop()](https://www.javatpoint.com/java-stack#pop) | E | The method removes an element from the top of the stack and returns the same  element as the value of that function. |
| [peek()](https://www.javatpoint.com/java-stack#peek) | E | The method looks at the top element of the stack without removing it. |
| [search(Object o)](https://www.javatpoint.com/java-stack#search) | int | The method searches the specified object and returns the position of the object. |

### **Stack Class empty() Method**

The **empty()** method of the Stack class check the stack is empty or not. If the stack is empty, it returns true, else returns false. We can also use the [isEmpty() method of the Vector class](https://www.javatpoint.com/java-vector-isempty-method).

**Syntax**

1. **public** **boolean** empty()

**Returns:** The method returns true if the stack is empty, else returns false.

In the following example, we have created an instance of the Stack class. After that, we have invoked the empty() method two times. The first time it returns **true** because we have not pushed any element into the stack. After that, we have pushed elements into the stack. Again we have invoked the empty() method that returns **false** because the stack is not empty.

**StackEmptyMethodExample.java**

1. **import** java.util.Stack;
2. **public** **class** StackEmptyMethodExample
3. {
4. **public** **static** **void** main(String[] args)
5. {
6. //creating an instance of Stack class
7. Stack<Integer> stk= **new** Stack<>();
8. // checking stack is empty or not
9. **boolean** result = stk.empty();
10. System.out.println("Is the stack empty? " + result);
11. // pushing elements into stack
12. stk.push(78);
13. stk.push(113);
14. stk.push(90);
15. stk.push(120);
16. //prints elements of the stack
17. System.out.println("Elements in Stack: " + stk);
18. result = stk.empty();
19. System.out.println("Is the stack empty? " + result);
20. }
21. }

**Output:**

Is the stack empty? true

Elements in Stack: [78, 113, 90, 120]

Is the stack empty? false

### **Stack Class push() Method**

The method inserts an item onto the top of the stack. It works the same as the method [addElement(item) method](https://www.javatpoint.com/java-vector-addelement-method) of the Vector class. It passes a parameter **item** to be pushed into the stack.

**Syntax**

1. **public** E push(E item)

**Parameter:** An item to be pushed onto the top of the stack.

**Returns:** The method returns the argument that we have passed as a parameter.

### **Stack Class pop() Method**

The method removes an object at the top of the stack and returns the same object. It throws **EmptyStackException** if the stack is empty.

**Syntax**

1. **public** E pop()

**Returns:** It returns an object that is at the top of the stack.

Let's implement the stack in a Java program and perform push and pop operations.

**StackPushPopExample.java**

### import java.util.Stack;

### class StackEmptyMethodExample

### {

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

### {

### //creating an instance of Stack class

### Stack<Integer> stk= new Stack<>();

### // checking stack is empty or not

### boolean result = stk.empty();

### System.out.println("Is the stack empty? " + result);

### // pushing elements into stack

### stk.push(78);

### stk.push(113);

### stk.push(90);

### stk.push(120);

### //prints elements of the stack

### System.out.println("Elements in Stack: " + stk);

### result = stk.empty();

### System.out.println("Is the stack empty? " + result);

### System.out.println(stk.pop());

### System.out.println("Elements in Stack: " + stk);

### }

### }

|  |
| --- |
| import java.util.\*;    public class StackDemo {      public static void main(String args[])      {          // Creating an empty Stack          Stack<Integer> STACK = new Stack<Integer>();            // Use add() method to add elements          STACK.push(10);          STACK.push(15);          STACK.push(30);          STACK.push(20);          STACK.push(5);            // Displaying the Stack          System.out.println("Initial Stack: " + STACK);            // Removing elements using pop() method          System.out.println("Popped element: " +                                           STACK.pop());          System.out.println("Popped element: " +                                           STACK.pop());            // Displaying the Stack after pop operation          System.out.println("Stack after pop operation "                                               + STACK);      }  } |

**Output:**

Initial Stack: [10, 15, 30, 20, 5]

Popped element: 5

Popped element: 20

Stack after pop operation [10, 15, 30]

### **Stack Class peek() Method**

It looks at the element that is at the top in the stack. It also throws **EmptyStackException** if the stack is empty.

**Syntax**

1. **public** E peek()

**Returns:** It returns the top elements of the stack.

Let's see an example of the peek() method.

**StackPeekMethodExample.java**

1. **import** java.util.Stack;
2. **public** **class** StackPeekMethodExample
3. {
4. **public** **static** **void** main(String[] args)
5. {
6. Stack<String> stk= **new** Stack<>();
7. // pushing elements into Stack
8. stk.push("Apple");
9. stk.push("Grapes");
10. stk.push("Mango");
11. stk.push("Orange");
12. System.out.println("Stack: " + stk);
13. // Access element from the top of the stack
14. String fruits = stk.peek();
15. //prints stack
16. System.out.println("Element at top: " + fruits);
17. }
18. }

**Output:**

Stack: [Apple, Grapes, Mango, Orange]

Element at the top of the stack: Orange

|  |
| --- |
| import java.util.\*;    public class StackDemo {      public static void main(String args[])      {          // Creating an empty Stack          Stack<Integer> STACK = new Stack<Integer>();            // Use push() to add elements into the Stack          STACK.push(10);          STACK.push(15);          STACK.push(30);          STACK.push(20);          STACK.push(5);            // Displaying the Stack          System.out.println("Initial Stack: " + STACK);            // Fetching the element at the head of the Stack          System.out.println("The element at the top of the"                             + " stack is: " + STACK.peek());            // Displaying the Stack after the Operation          System.out.println("Final Stack: " + STACK);      }  } |

**Output:**

Initial Stack: [10, 15, 30, 20, 5]

The element at the top of the stack is: 5

Final Stack: [10, 15, 30, 20, 5]

### **Stack Class search() Method**

The method searches the object in the stack from the top. It parses a parameter that we want to search for. It returns the 1-based location of the object in the stack. Thes topmost object of the stack is considered at distance 1.

Suppose, o is an object in the stack that we want to search for. The method returns the distance from the top of the stack of the occurrence nearest the top of the stack. It uses **equals()** method to search an object in the stack.

**Syntax**

1. **public** **int** search(Object o)

**Parameter:** o is the desired object to be searched.

**Returns:** It returns the object location from the top of the stack. If it returns -1, it means that the object is not on the stack.

Let's see an example of the search() method.

**StackSearchMethodExample.java**

1. **import** java.util.Stack;
2. **public** **class** StackSearchMethodExample
3. {
4. **public** **static** **void** main(String[] args)
5. {
6. Stack<String> stk= **new** Stack<>();
7. //pushing elements into Stack
8. stk.push("Mac Book");
9. stk.push("HP");
10. stk.push("DELL");
11. stk.push("Asus");
12. System.out.println("Stack: " + stk);
13. // Search an element
14. **int** location = stk.search("HP");
15. System.out.println("Location of Dell: " + location);
16. }
17. }

# ConcurrentModificationException in Java

The ConcurrentModificationException occurs when an object is tried to be modified concurrently when it is not permissible. This exception usually comes when one is working with **Java Collection classes**.

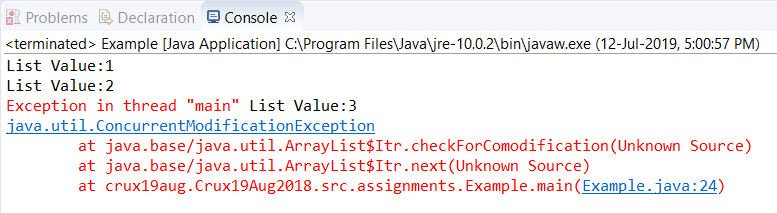
**For Example** - It is not permissible for a thread to modify a Collection when some other thread is iterating over it. This is because the result of the iteration becomes undefined with it. Some implementation of the Iterator class throws this exception, including all those general-purpose implementations of Iterator which are provided by the JRE. Iterators which do this are called **fail-fast** as they throw the exception quickly as soon as they encounter such situation rather than facing undetermined behavior of the collection any time in the future.

#### Note:**It is not mandatory that this exception will be thrown only when some other thread tries to modify a Collection object. It can also happen if a single thread has some methods called which are trying to violate the contract of the object. This may happen when a thread is trying to modify the Collection object while it is being iterated by some**fail-fast iterator**, the iterator will throw the exception.**

### **Example**

1. **import** java.awt.List;
2. **import** java.util.\*;
4. **public** **class** Concurrentmodificationexception {
6. **public** **static** **void** main(String[] args) {
7. ArrayList<Integer> list = **new** ArrayList<>();
9. list.add(1);
10. list.add(2);
11. list.add(3);
12. list.add(4);
13. list.add(5);
15. Iterator<Integer> it = list.iterator();
16. **while** (it.hasNext()) {
17. Integer value = it.next();
18. System.out.println("List Value:" + value);
19. **if** (value.equals(3))
20. list.remove(value);
21. }
23. }
25. }

**Output:**



This message says that the exception is thrown when the next method is called as the iterator is iterating the list and we are making modifications in it simultaneously. But if we make modifications in hashmap like given below, then it will not throw any such exception as the size of the hashmap won't change.

**For Example-**

1. **import** java.awt.List;
2. **import** java.util.\*;
4. **public** **class** concurrentmodificationexception {
6. **public** **static** **void** main(String[] args) {
8. HashMap<Integer, Integer> map = **new** HashMap<>();
9. map.put(1, 1);
10. map.put(2, 2);
11. map.put(3,3);
13. Iterator<Integer> it = map.keySet().iterator();
14. **while**(it.hasNext()) {
15. Integer key = it.next();
16. System.out.println("Map Value:" + map.get(key));
17. **if** (key.equals(2)) {
18. map.put(1, 4);
19. }
20. }
21. }
22. }

**Output:**

Map Value:1

Map Value:2

Map Value:3

This example works completely fine as while the iterator is iterating over the map, the size of the map is not changing. Only the map is being updated in the **if statement**.

### **Difference between Array List, Linked List, and Vector:**

| **Subject** | **Array List** | **Linked List** | **Vector** |
| --- | --- | --- | --- |
| **synchronized** | Not present | Not present | present |
| **Random access** | Allowed | Not Allowed | Allowed |
| **Memory Location** | contiguous  (continuous memory allocation) | Not contiguous | contiguous |
| **Null values** | supports | supports | supports |
| **Data structure** | Dynamic Array | Doubly Linked List | Dynamic Array |
| **Duplicate allowed** | Yes | Yes | Yes |
| **Operation** | Insertion and deletion are slow | Insertion and deletion are fast | Insertion and deletion are slow |

## **What is a dynamic array?**

The dynamic array is a **variable size** list data structure. It grows automatically when we try to insert an element if there is no more space left for the new element. It allows us to add and remove elements. It allocates memory at run time using the heap. It can change its size during run time.

### **Which one is better among Linked list, Array list, or Vector?**

It depends on the specific use case, each of these data structures has its own advantages and trade-offs. If you mostly need to insert and delete elements at the start or middle of the container, then a linked list might be a better option. If you need fast random access and are willing to accept slower insertion and deletion at end positions, an Array List or Vector is a better option.

# Java HashSet



Java HashSet class is 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.

### **HashSet class declaration**

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

1. **public** **class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>, Cloneable, Serializable

## **Constructors of Java HashSet class**

|  |  |  |
| --- | --- | --- |
| **SN** | **Constructor** | **Description** |
| 1) | HashSet() | It is used to construct a default HashSet. |
| 2) | HashSet(int capacity) | It is used to initialize the capacity of the hash set to the given integer value capacity.  The capacity grows automatically as elements are added to the HashSet. |
| 3) | HashSet(int capacity, float loadFactor) | It is used to initialize the capacity of the hash set to the given integer value capacity  and the specified load factor. |
| 4) | HashSet(Collection<? extends E> c) | It is used to initialize the hash set by using the elements of the collection c. |

## **Methods of Java HashSet class**

Various methods of Java HashSet class are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Modifier & Type** | **Method** | **Description** |
| 1) | boolean | [add(E e)](https://www.javatpoint.com/java-hashset-add-method) | It is used to add the specified element to this set if it is not already present. |
| 2) | void | [clear()](https://www.javatpoint.com/java-hashset-clear-method) | It is used to remove all of the elements from the set. |
| 3) | object | [clone()](https://www.javatpoint.com/java-hashset-clone-method) | It is used to return a shallow copy of this HashSet instance: the elements  themselves are not cloned. |
| 4) | boolean | [contains(Object o)](https://www.javatpoint.com/java-hashset-contains-method) | It is used to return true if this set contains the specified element. |
| 5) | boolean | [isEmpty()](https://www.javatpoint.com/java-hashset-isempty-method) | It is used to return true if this set contains no elements. |
| 6) | Iterator<E> | [iterator()](https://www.javatpoint.com/java-hashset-iterator-method) | It is used to return an iterator over the elements in this set. |
| 7) | boolean | [remove(Object o)](https://www.javatpoint.com/java-hashset-remove-method) | It is used to remove the specified element from this set if it is present. |
| 8) | int | [size()](https://www.javatpoint.com/java-hashset-size-method) | It is used to return the number of elements in the set. |
| 9) | Spliterator<E> | [spliterator()](https://www.javatpoint.com/java-hashset-spliterator-method) | It is used to create a late-binding and fail-fast Spliterator over the elements  in the set. |

### **Java HashSet Example**

Let's see a simple example of HashSet. Notice, the elements iterate in an unordered collection.

1. **import** java.util.\*;
2. **class** HashSet1{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet();
6. set.add("One");
7. set.add("Two");
8. set.add("Three");
9. set.add("Four");
10. set.add("Five");
11. Iterator<String> i=set.iterator();
12. **while**(i.hasNext())
13. {
14. System.out.println(i.next());
15. }
16. }
17. }

Five

One

Four

Two

Three

### **Java HashSet example ignoring duplicate elements**

In this example, we see that HashSet doesn't allow duplicate elements.

1. **import** java.util.\*;
2. **class** HashSet2{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. HashSet<String> set=**new** HashSet<String>();
6. set.add("Ravi");
7. set.add("Vijay");
8. set.add("Ravi");
9. set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

Ajay

Vijay

Ravi

### **Java HashSet example to remove elements**

Here, we see different ways to remove an element.

1. **import** java.util.\*;
2. **class** HashSet3{
3. **public** **static** **void** main(String args[]){
4. HashSet<String> set=**new** HashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Arun");
8. set.add("Sumit");
9. System.out.println("An initial list of elements: "+set);
10. //Removing specific element from HashSet
11. set.remove("Ravi");
12. System.out.println("After invoking remove(object) method: "+set);
13. HashSet<String> set1=**new** HashSet<String>();
14. set1.add("Ajay");
15. set1.add("Gaurav");
16. set.addAll(set1);
17. System.out.println("Updated List: "+set);
18. //Removing all the new elements from HashSet
19. set.removeAll(set1);
20. System.out.println("After invoking removeAll() method: "+set);
21. //Removing elements on the basis of specified condition
22. set.removeIf(str->str.contains("Vijay"));
23. System.out.println("After invoking removeIf() method: "+set);
24. //Removing all the elements available in the set
25. set.clear();
26. System.out.println("After invoking clear() method: "+set);
27. }
28. }

An initial list of elements: [Vijay, Ravi, Arun, Sumit]

After invoking remove(object) method: [Vijay, Arun, Sumit]

Updated List: [Vijay, Arun, Gaurav, Sumit, Ajay]

After invoking removeAll() method: [Vijay, Arun, Sumit]

After invoking removeIf() method: [Arun, Sumit]

After invoking clear() method: []

### **Java HashSet from another Collection**

1. **import** java.util.\*;
2. **class** HashSet4{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> list=**new** ArrayList<String>();
5. list.add("Ravi");
6. list.add("Vijay");
7. list.add("Ajay");
9. HashSet<String> set=**new** HashSet(list);
10. set.add("Gaurav");
11. Iterator<String> i=set.iterator();
12. **while**(i.hasNext())
13. {
14. System.out.println(i.next());
15. }
16. }
17. }

Vijay

Ravi

Gaurav

Ajay

### **Java HashSet Example: Book**

Let's see a HashSet example where we are adding books to set and printing all the books.

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** HashSetExample {
15. **public** **static** **void** main(String[] args) {
16. HashSet<Book> set=**new** HashSet<Book>();
17. //Creating Books
18. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
21. //Adding Books to HashSet
22. set.add(b1);
23. set.add(b2);
24. set.add(b3);
25. //Traversing HashSet
26. **for**(Book b:set){
27. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28. }
29. }
30. }

Output:

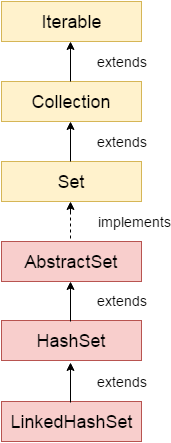
101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

| **Sr. No.** | **Key** | **ArrayList** | **HashSet** |
| --- | --- | --- | --- |
| 1 | Implementation | ArrayList is the implementation of the list interface. | HashSet on the other hand is the implementation of a  set interface. |
| 2 | Internal implementation | ArrayList internally implements array for its implementation. | HashSet internally uses Hashmap for its implementation. |
| 3 | Order of elements | ArrayList maintains the insertion order i.e order of the object  in which they are inserted. | HashSet is an unordered collection and doesn't  maintain any order. |
| 4 | Duplicates | ArrayList allows duplicate values in its collection. | On other hand duplicate elements are not allowed in  Hashset. |
| 5 | Index performance | ArrayList uses index for its performance i.e its index based one can retrieve object by calling get(index) or remove objects by calling remove(index) | HashSet is completely based on object also it doesn't  provide get() method. |
| 6 | Null Allowed | Any number of null value can be inserted in arraylist without any restriction. | On other hand Hashset allows only one null value in its  collection,after which no null value is allowed to be  added. |

# Java LinkedHashSet Class



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

The important points about the Java LinkedHashSet class are:

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

#### **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.**

### **Constructors of Java LinkedHashSet Class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashSet() | It is used to construct a default HashSet. |
| HashSet(Collection c) | It is used to initialize the hash set by using the elements of the collection c. |
| LinkedHashSet(int capacity) | It is used to initialize the capacity of the linked hash set to the given integer value capacity. |
| LinkedHashSet(int capacity, float fillRatio) | It is used to initialize both the capacity and the fill ratio (also called load capacity) of the  hash set from its argument. |

### **Java LinkedHashSet Example**

Let's see a simple example of the Java LinkedHashSet class. Here you can notice that the elements iterate in insertion order.

**FileName:** LinkedHashSet1.java

1. **import** java.util.\*;
2. **class** LinkedHashSet1{
3. **public** **static** **void** main(String args[]){
4. //Creating HashSet and adding elements
5. LinkedHashSet<String> set=**new** LinkedHashSet();
6. set.add("One");
7. set.add("Two");
8. set.add("Three");
9. set.add("Four");
10. set.add("Five");
11. Iterator<String> i=set.iterator();
12. **while**(i.hasNext())
13. {
14. System.out.println(i.next());
15. }
16. }
17. }

**Output:**

One

Two

Three

Four

Five

#### **Note: We can also use the enhanced for loop for displaying the elements.**

### **Java LinkedHashSet example ignoring duplicate Elements**

**sFileName:** LinkedHashSet2.java

1. **import** java.util.\*;
2. **class** LinkedHashSet2{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> al=**new** LinkedHashSet<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. **while**(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }

**Output:**

Ravi

Vijay

Ajay

### **Remove Elements Using LinkeHashSet Class**

1. **import** java.util.\*;
3. **public** **class** LinkedHashSet3
4. {
6. // main method
7. **public** **static** **void** main(String argvs[])
8. {
10. // Creating an empty LinekdhashSet of string type
11. LinkedHashSet<String> lhs = **new** LinkedHashSet<String>();
13. // Adding elements to the above Set
14. // by invoking the add() method
15. lhs.add("Java");
16. lhs.add("T");
17. lhs.add("Point");
18. lhs.add("Good");
19. lhs.add("Website");
21. // displaying all the elements on the console
22. System.out.println("The hash set is: " + lhs);
24. // Removing an element from the above linked Set
26. // since the element "Good" is present, therefore, the method remove()
27. // returns true
28. System.out.println(lhs.remove("Good"));
30. // After removing the element
31. System.out.println("After removing the element, the hash set is: " + lhs);
33. // since the element "For" is not present, therefore, the method remove()
34. // returns false
35. System.out.println(lhs.remove("For"));
37. }
38. }  **Output:**

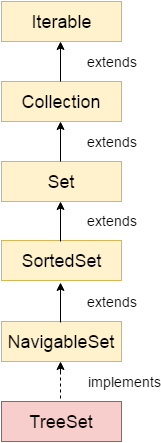
The hash set is: [Java, T, Point, Good, Website]

true

After removing the element, the hash set is: [Java, T, Point, Website]

false

# 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 the 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.
* Java TreeSet class contains unique elements only like HashSet.
* Java TreeSet class access and retrieval times are quite fast.
* Java TreeSet class doesn't allow null elements.
* Java TreeSet class is non-synchronized.
* Java TreeSet class maintains ascending order.
* The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.

### **Internal Working of The TreeSet Class**

TreeSet is being implemented using a binary search tree, which is self-balancing just like a Red-Black Tree. Therefore, operations such as a search, remove, and add consume O(log(N)) time. The reason behind this is there in the self-balancing tree. It is there to ensure that the tree height never exceeds O(log(N)) for all of the mentioned operations. Therefore, it is one of the efficient data structures in order to keep the large data that is sorted and also to do operations on it.

### **Synchronization of The TreeSet Class**

As already mentioned above, the TreeSet class is not synchronized. It means if more than one thread concurrently accesses a tree set, and one of the accessing threads modify it, then the synchronization must be done manually. It is usually done by doing some object synchronization that encapsulates the set. However, in the case where no such object is found, then the set must be wrapped with the help of the Collections.synchronizedSet() method. It is advised to use the method during creation time in order to avoid the unsynchronized access of the set. The following code snippet shows the same.

### **Constructors of Java TreeSet Class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| TreeSet() | It is used to construct an empty tree set that will be sorted in ascending order  according to the natural order of the tree set. |
| TreeSet(Collection<? extends E> c) | It is used to build a new tree set that contains the elements of the collection c. |
| TreeSet(Comparator<? super E> comparator) | It is used to construct an empty tree set that will be sorted according to given  comparator. |
| TreeSet(SortedSet<E> s) | It is used to build a TreeSet that contains the elements of the given SortedSet. |

### **Methods of Java TreeSet Class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(E e) | It is used to add the specified element to this set if it is not already present. |
| boolean addAll(Collection<? extends E> c) | It is used to add all of the elements in the specified collection to this set. |
| E ceiling(E e) | It returns the equal or closest greatest element of the specified element from the set, or null there is no such element. |
| Comparator<? super E> comparator() | It returns a comparator that arranges elements in order. |
| Iterator descendingIterator() | It is used to iterate the elements in descending order. |
| NavigableSet descendingSet() | It returns the elements in reverse order. |
| E floor(E e) | It returns the equal or closest least element of the specified element from the  set, or null there is no such element. |
| SortedSet headSet(E toElement) | It returns the group of elements that are less than the specified element. |
| NavigableSet headSet(E toElement, boolean inclusive) | It returns the group of elements that are less than or equal to(if, inclusive is  true) the specified element. |
| E higher(E e) | It returns the closest greatest element of the specified element from the set,  or null there is no such element. |
| Iterator iterator() | It is used to iterate the elements in ascending order. |
| E lower(E e) | It returns the closest least element of the specified element from the set,  or null there is no such element. |
| E pollFirst() | It is used to retrieve and remove the lowest(first) element. |
| E pollLast() | It is used to retrieve and remove the highest(last) element. |
| Spliterator spliterator() | It is used to create a late-binding and fail-fast spliterator over the elements. |
| NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) | It returns a set of elements that lie between the given range. |
| SortedSet subSet(E fromElement, E toElement)) | It returns a set of elements that lie between the given range which includes  fromElement and excludes toElement. |
| SortedSet tailSet(E fromElement) | It returns a set of elements that are greater than or equal to the specified  element. |
| NavigableSet tailSet(E fromElement, boolean inclusive) | It returns a set of elements that are greater than or equal to (if, inclusive is  true) the specified element. |
| boolean contains(Object o) | It returns true if this set contains the specified element. |
| boolean isEmpty() | It returns true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| void clear() | It is used to remove all of the elements from this set. |
| Object clone() | It returns a shallow copy of this TreeSet instance. |
| E first() | It returns the first (lowest) element currently in this sorted set. |
| E last() | It returns the last (highest) element currently in this sorted set. |
| int size() | It returns the number of elements in this set. |

### **Java TreeSet Examples**

1. **import** java.util.\*;
2. **class** TreeSet1{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> al=**new** TreeSet<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

**Output:**

Ajay

Ravi

Vijay

### **Java TreeSet Example 2:**

Let's see an example of traversing elements in descending order.

1. **import** java.util.\*;
2. **class** TreeSet2{
3. **public** **static** **void** main(String args[]){
4. TreeSet<String> set=**new** TreeSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Ajay");
8. System.out.println("Traversing element through Iterator in descending order");
9. Iterator i=set.descendingIterator();
10. **while**(i.hasNext())
11. {
12. System.out.println(i.next());
13. }
15. }
16. }

**Output:**

Traversing element through Iterator in descending order

Vijay

Ravi

Ajay

Traversing element through NavigableSet in descending order

Vijay

Ravi

Ajay

### **Java TreeSet Example 3:**

Let's see an example to retrieve and remove the highest and lowest Value.

**FileName:** TreeSet3.java

1. **import** java.util.\*;
2. **class** TreeSet3{
3. **public** **static** **void** main(String args[]){
4. TreeSet<Integer> set=**new** TreeSet<Integer>();
5. set.add(24);
6. set.add(66);
7. set.add(12);
8. set.add(15);
9. System.out.println("Lowest Value: "+set.pollFirst());
10. System.out.println("Highest Value: "+set.pollLast());
11. }
12. }

**Output:**

Lowest Value: 12

Highest Value: 66

### **Java TreeSet Example 4:**

In this example, we perform various NavigableSet operations.

1. **import** java.util.\*;
2. **class** TreeSet4{
3. **public** **static** **void** main(String args[]){
4. TreeSet<String> set=**new** TreeSet<String>();
5. set.add("A");
6. set.add("B");
7. set.add("C");
8. set.add("D");
9. set.add("E");
10. System.out.println("Initial Set: "+set);
12. System.out.println("Reverse Set: "+set.descendingSet());
14. System.out.println("Head Set: "+set.headSet("C", **true**));
16. System.out.println("SubSet: "+set.subSet("A", **false**, "E", **true**));
18. System.out.println("TailSet: "+set.tailSet("C", **false**));
19. }
20. }

**Output:**

Initial Set: [A, B, C, D, E]

Reverse Set: [E, D, C, B, A]

Head Set: [A, B, C]

SubSet: [B, C, D, E]

TailSet: [D, E]

### **Java TreeSet Example 5:**

In this example, we perform various SortedSetSet operations.

1. **import** java.util.\*;
2. **class** TreeSet5{
3. **public** **static** **void** main(String args[]){
4. TreeSet<String> set=**new** TreeSet<String>();
5. set.add("A");
6. set.add("B");
7. set.add("C");
8. set.add("D");
9. set.add("E");
11. System.out.println("Intial Set: "+set);
13. System.out.println("Head Set: "+set.headSet("C"));
15. System.out.println("SubSet: "+set.subSet("A", "E"));
17. System.out.println("TailSet: "+set.tailSet("C"));
18. }
19. }

**Output:**

Intial Set: [A, B, C, D, E]

Head Set: [A, B]

SubSet: [A, B, C, D]

TailSet: [C, D, E]

**Differences Between HashSet, LinkedHashSet**,**and TreeSet:**

| **Features** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| --- | --- | --- | --- |
| Internal Working | HashSet internally uses HashMap for storing objects | LinkedHashSet uses LinkedHashMap internally to store objects | TreeSet uses TreeMap internally to store objects |
| When To Use | If you don’t want to maintain insertion order but want to store unique objects | If you want to maintain the insertion order of elements then you can use LinkedHashSet | If you want to sort the elements according to some Comparator then use TreeSet |
| Order | HashSet does not maintain insertion order | LinkedHashSet maintains the insertion order of objects | While TreeSet orders the elements according to supplied Comparator. By default, objects will be placed according to their natural ascending order. |
| Complexity of Operations | HashSet gives O(1) complexity for insertion, removing, and retrieving objects | LinkedHashSet gives insertion, removing, and retrieving operations performance in order O(1). | While TreeSet gives the performance of order O(log(n)) for insertion, removing, and retrieving operations. |
| Performance | The performance of HashSet is better when compared to LinkedHashSet and TreeSet. | The performance of LinkedHashSet is slower than TreeSet. It is almost similar to HashSet but slower because LinkedHashSet internally maintains LinkedList to maintain the insertion order of elements | TreeSet performance is better than LinkedHashSet except for insertion and removal operations because it has to sort the elements after each insertion and removal operation. |
| Compare | HashSet uses equals() and hashCode() methods to compare the objects | LinkedHashSet uses equals() and hashCode() methods to compare it’s objects | TreeSet uses compare() and compareTo() methods to compare the objects |
| Null Elements | HashSet allows only one null value. | LinkedHashSet allows only one null value. | TreeSet does not permit null value. If you insert null value into TreeSet, it will throw NullPointerException. |
| Syntax | HashSet obj = new HashSet(); | LinkedHashSet obj = new LinkedHashSet(); | TreeSet obj = new TreeSet(); |

**Similarities Between HashSet, LinkedHashSet**,**and TreeSet:**

* Duplicates: HashSet, LinkedHashSet and TreeSet are implements Set interface, so they are not allowed to store duplicates objects.
* Thread-safe: If we want to use HashSet, LinkedHashSet, and TreeSet in a multi-threading environment then first we make it externally synchronized because both LinkedHashSet and TreeSet are not thread-safe.
* All three are Cloneable and Serializable.

**When to use HashSet, TreeSet**,**and LinkedHashSet in Java:**

1. **HashSet:** If you don’t want to maintain insertion order but want to store unique objects.
2. **LinkedHashSet:** If you want to maintain the insertion order of elements then you can use LinkedHashSet.
3. **TreeSet:** If you want to sort the elements according to some Comparator then use TreeSet.

So as you see the output of the above program according to that and according to your requirements, you can choose anyone from HashSet, TreeSet, and LinkedHashSet.

# Java Queue Interface

The interface Queue is available in the java.util package and does extend the Collection interface. It is used to keep the elements that are processed in the First In First Out (FIFO) manner. It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list.

Being an interface, the queue requires, for the declaration, a concrete class, and the most common classes are the LinkedList and PriorityQueue in Java. Implementations done by these classes are not thread safe. If it is required to have a thread safe implementation, PriorityBlockingQueue is an available option.

### **Queue Interface Declaration**

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

### **Methods of Java Queue Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(object) | It is used to insert the specified element into this queue and return true upon success. |
| boolean offer(object) | It is used to insert the specified element into this queue. |
| Object remove() | It is used to retrieves and removes the head of this queue. |
| Object poll() | It is used to retrieves and removes the head of this queue, or returns null if this queue is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this queue. |
| Object peek() | It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is  empty. |

## **Features of a Queue**

The following are some important features of a queue.

* As discussed earlier, FIFO concept is used for insertion and deletion of elements from a queue.
* The Java Queue provides support for all of the methods of the Collection interface including deletion, insertion, etc.
* PriorityQueue, ArrayBlockingQueue and LinkedList are the implementations that are used most frequently.
* The NullPointerException is raised, if any null operation is done on the BlockingQueues.
* Those Queues that are present in the util package are known as Unbounded Queues.
* Those Queues that are present in the util.concurrent package are known as bounded Queues.
* All Queues barring the Deques facilitates removal and insertion at the head and tail of the queue; respectively. In fact, deques support element insertion and removal at both ends.

## **PriorityQueue Class**

PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority. It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue. However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.

### **PriorityQueue Class Declaration**

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

1. **public** **class** PriorityQueue<E> **extends** AbstractQueue<E> **implements** Serializable

### **Java PriorityQueue Example**

**FileName:** TestCollection12.java

1. **import** java.util.\*;
2. **class** TestCollection12{
3. **public** **static** **void** main(String args[]){
4. PriorityQueue<String> queue=**new** PriorityQueue<String>();
5. queue.add("Amit");
6. queue.add("Vijay");
7. queue.add("Karan");
8. queue.add("Jai");
9. queue.add("Rahul");
10. System.out.println("head:"+queue.element());
11. System.out.println("head:"+queue.peek());
12. System.out.println("iterating the queue elements:");
13. Iterator itr=queue.iterator();
14. **while**(itr.hasNext()){
15. System.out.println(itr.next());
16. }
17. queue.remove();
18. queue.poll();
19. System.out.println("after removing two elements:");
20. Iterator<String> itr2=queue.iterator();
21. **while**(itr2.hasNext()){
22. System.out.println(itr2.next());
23. }
24. }
25. }

**Output:**

head:Amit

head:Amit

iterating the queue elements:

Amit

Jai

Karan

Vijay

Rahul

after removing two elements:

Karan

Rahul

Vijay

### **Java PriorityQueue Example: Book**

Let's see a PriorityQueue example where we are adding books to queue and printing all the books. The elements in PriorityQueue must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in PriorityQueue, you need to implement Comparable interface.

**FileName:** LinkedListExample.java

1. **import** java.util.\*;
2. **class** Book **implements** Comparable<Book>{
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. **public** **int** compareTo(Book b) {
14. **if**(id>b.id){
15. **return** 1;
16. }**else** **if**(id<b.id){
17. **return** -1;
18. }**else**{
19. **return** 0;
20. }
21. }
22. }
23. **public** **class** LinkedListExample {
24. **public** **static** **void** main(String[] args) {
25. Queue<Book> queue=**new** PriorityQueue<Book>();
26. //Creating Books
27. Book b1=**new** Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
28. Book b2=**new** Book(233,"Operating System","Galvin","Wiley",6);
29. Book b3=**new** Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
30. //Adding Books to the queue
31. queue.add(b1);
32. queue.add(b2);
33. queue.add(b3);
34. System.out.println("Traversing the queue elements:");
35. //Traversing queue elements
36. **for**(Book b:queue){
37. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
38. }
39. queue.remove();
40. System.out.println("After removing one book record:");
41. **for**(Book b:queue){
42. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
43. }
44. }
45. }

**Output:**

Traversing the queue elements:

101 Data Communications & Networking Forouzan Mc Graw Hill 4

233 Operating System Galvin Wiley 6

121 Let us C Yashwant Kanetkar BPB 8

After removing one book record:

121 Let us C Yashwant Kanetkar BPB 8

233 Operating System Galvin Wiley 6

# Java Deque Interface

The interface called Deque is present in java.util package. It is the subtype of the interface queue. The Deque supports the addition as well as the removal of elements from both ends of the data structure. Therefore, a deque can be used as a stack or a queue. We know that the stack supports the Last In First Out (LIFO) operation, and the operation First In First Out is supported by a queue. As a deque supports both, either of the mentioned operations can be performed on it. Deque is an acronym for **"double ended queue".**

## **Deque Interface declaration**

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

### **Methods of Java Deque Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(object) | It is used to insert the specified element into this deque and return true upon success. |
| boolean offer(object) | It is used to insert the specified element into this deque. |
| Object remove() | It is used to retrieve and removes the head of this deque. |
| Object poll() | It is used to retrieve and removes the head of this deque, or returns null if this deque is empty. |
| Object element() | It is used to retrieve, but does not remove, the head of this deque. |
| Object peek() | It is used to retrieve, but does not remove, the head of this deque, or returns null if this deque is  empty. |
| Object peekFirst() | The method returns the head element of the deque. The method does not remove any element  from the deque. Null is returned by this method, when the deque is empty. |
| Object peekLast() | The method returns the last element of the deque. The method does not remove any element  from the deque. Null is returned by this method, when the deque is empty. |
| Boolean offerFirst(e) | Inserts the element e at the front of the queue. If the insertion is successful, true is returned;  otherwise, false. |
| Object offerLast(e) | Inserts the element e at the tail of the queue. If the insertion is successful, true is returned;  otherwise, false. |

java arraydeque hierarchy

## **ArrayDeque class**

We know that it is not possible to create an object of an interface in Java. Therefore, for instantiation, we need a class that implements the Deque interface, and that class is ArrayDeque. It grows and shrinks as per usage. It also inherits the AbstractCollection class.

The important points about ArrayDeque class are:

* Unlike Queue, we can add or remove elements from both sides.
* Null elements are not allowed in the ArrayDeque.
* ArrayDeque is not thread safe, in the absence of external synchronization.
* ArrayDeque has no capacity restrictions.
* ArrayDeque is faster than LinkedList and Stack.

### **ArrayDeque Hierarchy**

The hierarchy of ArrayDeque class is given in the figure displayed at the right side of the page.

### **ArrayDeque class declaration**

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

1. **public** **class** ArrayDeque<E> **extends** AbstractCollection<E> **implements** Deque<E>, Cloneable, Serializable

## **Java ArrayDeque Example**

**FileName:** ArrayDequeExample.java

1. **import** java.util.\*;
2. **public** **class** ArrayDequeExample {
3. **public** **static** **void** main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = **new** ArrayDeque<String>();
6. deque.add("Ravi");
7. deque.add("Vijay");
8. deque.add("Ajay");
9. //Traversing elements
10. **for** (String str : deque) {
11. System.out.println(str);
12. }
13. }
14. }

**Output:**

Ravi

Vijay

Ajay

## **Java ArrayDeque Example: offerFirst() and pollLast()**

**FileName:** DequeExample.java

1. **import** java.util.\*;
2. **public** **class** DequeExample {
3. **public** **static** **void** main(String[] args) {
4. Deque<String> deque=**new** ArrayDeque<String>();
5. deque.offer("arvind");
6. deque.offer("vimal");
7. deque.add("mukul");
8. deque.offerFirst("jai");
9. System.out.println("After offerFirst Traversal...");
10. **for**(String s:deque){
11. System.out.println(s);
12. }
13. //deque.poll();
14. //deque.pollFirst();//it is same as poll()
15. deque.pollLast();
16. System.out.println("After pollLast() Traversal...");
17. **for**(String s:deque){
18. System.out.println(s);
19. }
20. }
21. }

**Output:**

After offerFirst Traversal...

jai

arvind

vimal

mukul

After pollLast() Traversal...

jai

arvind

vimal

## **Java ArrayDeque Example: Book**

**FileName:** ArrayDequeExample.java

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ArrayDequeExample {
15. **public** **static** **void** main(String[] args) {
16. Deque<Book> set=**new** ArrayDeque<Book>();
17. //Creating Books
18. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
20. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
21. //Adding Books to Deque
22. set.add(b1);
23. set.add(b2);
24. set.add(b3);
25. //Traversing ArrayDeque
26. **for**(Book b:set){
27. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28. }
29. }
30. }

**Output:**

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

# 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:

Java 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: Non-Generic (Old Style)**

1. //Non-generic
2. **import** java.util.\*;
3. **public** **class** MapExample1 {
4. **public** **static** **void** main(String[] args) {
5. Map map=**new** HashMap();
6. //Adding elements to map
7. map.put(1,"Amit");
8. map.put(5,"Rahul");
9. map.put(2,"Jai");
10. map.put(6,"Amit");
11. //Traversing Map
12. Set set=map.entrySet();//Converting to Set so that we can traverse
13. Iterator itr=set.iterator();
14. **while**(itr.hasNext()){
15. //Converting to Map.Entry so that we can get key and value separately
16. Map.Entry entry=(Map.Entry)itr.next();
17. System.out.println(entry.getKey()+" "+entry.getValue());
18. }
19. }
20. }

Output:

1 Amit

2 Jai

5 Rahul

6 Amit

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

1. **import** java.util.\*;
2. **class** MapExample2{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. //Elements can traverse in any order
9. **for**(Map.Entry m:map.entrySet()){
10. System.out.println(m.getKey()+" "+m.getValue());
11. }
12. }
13. }

Output:

102 Rahul

100 Amit

101 Vijay

### **Java Map Example: comparingByKey()**

1. **import** java.util.\*;
2. **class** MapExample3{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. //Returns a Set view of the mappings contained in this map
9. map.entrySet()
10. //Returns a sequential Stream with this collection as its source
11. .stream()
12. //Sorted according to the provided Comparator
13. .sorted(Map.Entry.comparingByKey())
14. //Performs an action for each element of this stream
15. .forEach(System.out::println);
16. }
17. }

Output:

100=Amit

101=Vijay

102=Rahul

### **Java Map Example: comparingByKey() in Descending Order**

1. **import** java.util.\*;
2. **class** MapExample4{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. //Returns a Set view of the mappings contained in this map
9. map.entrySet()
10. //Returns a sequential Stream with this collection as its source
11. .stream()
12. //Sorted according to the provided Comparator
13. .sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))
14. //Performs an action for each element of this stream
15. .forEach(System.out::println);
16. }
17. }

Output:

102=Rahul

101=Vijay

100=Amit

### **Java Map Example: comparingByValue()**

1. **import** java.util.\*;
2. **class** MapExample5{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. //Returns a Set view of the mappings contained in this map
9. map.entrySet()
10. //Returns a sequential Stream with this collection as its source
11. .stream()
12. //Sorted according to the provided Comparator
13. .sorted(Map.Entry.comparingByValue())
14. //Performs an action for each element of this stream
15. .forEach(System.out::println);
16. }
17. }

Output:

100=Amit

102=Rahul

101=Vijay

### **Java Map Example: comparingByValue() in Descending Order**

1. **import** java.util.\*;
2. **class** MapExample6{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. //Returns a Set view of the mappings contained in this map
9. map.entrySet()
10. //Returns a sequential Stream with this collection as its source
11. .stream()
12. //Sorted according to the provided Comparator
13. .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
14. //Performs an action for each element of this stream
15. .forEach(System.out::println);
16. }
17. }

Output:

101=Vijay

102=Rahul

100=Amit

# Java HashMap



Java **HashMap** class implements the Map interface which allows us to store key and value pair, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

### **Points to remember**

* Java HashMap contains values based on the key.
* Java HashMap contains only unique keys.
* Java HashMap may have one null key and multiple null values.
* Java HashMap is non synchronized.
* Java HashMap 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 declaration**

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

Play Video

1. **public** **class** HashMap<K,V> **extends** AbstractMap<K,V> **implements** Map<K,V>, Cloneable, Serializable

### **HashMap class Parameters**

Let's see the Parameters for java.util.HashMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Constructors of Java HashMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| HashMap() | It is used to construct a default HashMap. |
| HashMap(Map<? extends K,? extends V> m) | It is used to initialize the hash map by using the elements of the given Map  object m. |
| HashMap(int capacity) | It is used to initializes the capacity of the hash map to the given integer value,  capacity. |
| HashMap(int capacity, float loadFactor) | It is used to initialize both the capacity and load factor of the hash map by  using its arguments. |

### **Methods of Java HashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to remove all of the mappings from this map. |
| boolean isEmpty() | It is used to return true if this map contains no key-value mappings. |
| Object clone() | It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| Set entrySet() | It is used to return a collection view of the mappings contained in this map. |
| Set keySet() | It is used to return a set view of the keys contained in this map. |
| 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. |
| 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. |
| 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<V> 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. |

### **Java HashMap Example**

Let's see a simple example of HashMap to store key and value pair.

1. **import** java.util.\*;
2. **public** **class** HashMapExample1{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> map=**new** HashMap<Integer,String>();//Creating HashMap
5. map.put(1,"Mango");  //Put elements in Map
6. map.put(2,"Apple");
7. map.put(3,"Banana");
8. map.put(4,"Grapes");
10. System.out.println("Iterating Hashmap...");
11. **for**(Map.Entry m : map.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }  Iterating Hashmap...

1 Mango

2 Apple

3 Banana

4 Grapes

In this example, we are storing Integer as the key and String as the value, so we are using HashMap<Integer,String> as the type. The put() method inserts the elements in the map.

To get the key and value elements, we should call the getKey() and getValue() methods. The Map.Entry interface contains the getKey() and getValue() methods. But, we should call the entrySet() method of Map interface to get the instance of Map.Entry.

### **No Duplicate Key on HashMap**

You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the value.

1. **import** java.util.\*;
2. **public** **class** HashMapExample2{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> map=**new** HashMap<Integer,String>();//Creating HashMap
5. map.put(1,"Mango");  //Put elements in Map
6. map.put(2,"Apple");
7. map.put(3,"Banana");
8. map.put(1,"Grapes"); //trying duplicate key
10. System.out.println("Iterating Hashmap...");
11. **for**(Map.Entry m : map.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

Iterating Hashmap...

1 Grapes

2 Apple

3 Banana

### **Java HashMap example to add() elements**

Here, we see different ways to insert elements.

1. **import** java.util.\*;
2. **class** HashMap1{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> hm=**new** HashMap<Integer,String>();
5. System.out.println("Initial list of elements: "+hm);
6. hm.put(100,"Amit");
7. hm.put(101,"Vijay");
8. hm.put(102,"Rahul");
10. System.out.println("After invoking put() method ");
11. **for**(Map.Entry m:hm.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
15. hm.putIfAbsent(103, "Gaurav");
16. System.out.println("After invoking putIfAbsent() method ");
17. **for**(Map.Entry m:hm.entrySet()){
18. System.out.println(m.getKey()+" "+m.getValue());
19. }
20. HashMap<Integer,String> map=**new** HashMap<Integer,String>();
21. map.put(104,"Ravi");
22. map.putAll(hm);
23. System.out.println("After invoking putAll() method ");
24. **for**(Map.Entry m:map.entrySet()){
25. System.out.println(m.getKey()+" "+m.getValue());
26. }
27. }
28. }

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 HashMap example to remove() elements**

Here, we see different ways to remove elements.

1. **import** java.util.\*;
2. **public** **class** HashMap2 {
3. **public** **static** **void** main(String args[]) {
4. HashMap<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. map.put(103, "Gaurav");
9. System.out.println("Initial list of elements: "+map);
10. //key-based removal
11. map.remove(100);
12. System.out.println("Updated list of elements: "+map);
13. //value-based removal
14. map.remove(101);
15. System.out.println("Updated list of elements: "+map);
16. //key-value pair based removal
17. map.remove(102, "Rahul");
18. System.out.println("Updated list of elements: "+map);
19. }
20. }

Output:

Initial list of elements: {100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {101=Vijay, 102=Rahul, 103=Gaurav}

Updated list of elements: {102=Rahul, 103=Gaurav}

Updated list of elements: {103=Gaurav}

### **Java HashMap example to replace() elements**

Here, we see different ways to replace elements.

1. **import** java.util.\*;
2. **class** HashMap3{
3. **public** **static** **void** main(String args[]){
4. HashMap<Integer,String> hm=**new** HashMap<Integer,String>();
5. hm.put(100,"Amit");
6. hm.put(101,"Vijay");
7. hm.put(102,"Rahul");
8. System.out.println("Initial list of elements:");
9. **for**(Map.Entry m:hm.entrySet())
10. {
11. System.out.println(m.getKey()+" "+m.getValue());
12. }
13. System.out.println("Updated list of elements:");
14. hm.replace(102, "Gaurav");
15. **for**(Map.Entry m:hm.entrySet())
16. {
17. System.out.println(m.getKey()+" "+m.getValue());
18. }
19. System.out.println("Updated list of elements:");
20. hm.replace(101, "Vijay", "Ravi");
21. **for**(Map.Entry m:hm.entrySet())
22. {
23. System.out.println(m.getKey()+" "+m.getValue());
24. }
25. System.out.println("Updated list of elements:");
26. hm.replaceAll((k,v) -> "Ajay");
27. **for**(Map.Entry m:hm.entrySet())
28. {
29. System.out.println(m.getKey()+" "+m.getValue());
30. }
31. }
32. }

Initial list of elements:

100 Amit

101 Vijay

102 Rahul

Updated list of elements:

100 Amit

101 Vijay

102 Gaurav

Updated list of elements:

100 Amit

101 Ravi

102 Gaurav

Updated list of elements:

100 Ajay

101 Ajay

102 Ajay

### **Difference between HashSet and HashMap**

HashSet contains only values whereas HashMap contains an entry(key and value).

# 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.

### **LinkedHashMap class declaration**

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

1. **public** **class** LinkedHashMap<K,V> **extends** HashMap<K,V> **implements** Map<K,V>

### **LinkedHashMap class Parameters**

Let's see the Parameters for java.util.LinkedHashMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Constructors of Java LinkedHashMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| LinkedHashMap() | It is used to construct a default LinkedHashMap. |
| LinkedHashMap(int capacity) | It is used to initialize a LinkedHashMap with the given capacity. |
| LinkedHashMap(int capacity, float loadFactor) | It is used to initialize both the capacity and the load factor. |
| LinkedHashMap(int capacity, float loadFactor, boolean accessOrder) | It is used to initialize both the capacity and the load factor with  specified ordering mode. |
| LinkedHashMap(Map<? extends K,? extends V> m) | It is used to initialize the LinkedHashMap with the elements from the  given Map class m. |

### **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**

1. **import** java.util.\*;
2. **class** LinkedHashMap1{
3. **public** **static** **void** main(String args[]){
5. LinkedHashMap<Integer,String> hm=**new** LinkedHashMap<Integer,String>();
7. hm.put(100,"Amit");
8. hm.put(101,"Vijay");
9. hm.put(102,"Rahul");
11. **for**(Map.Entry m:hm.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

Output:100 Amit

101 Vijay

102 Rahul

### **Java LinkedHashMap Example: Key-Value pair**

1. **import** java.util.\*;
2. **class** LinkedHashMap2{
3. **public** **static** **void** main(String args[]){
4. LinkedHashMap<Integer, String> map = **new** LinkedHashMap<Integer, String>();
5. map.put(100,"Amit");
6. map.put(101,"Vijay");
7. map.put(102,"Rahul");
8. //Fetching key
9. System.out.println("Keys: "+map.keySet());
10. //Fetching value
11. System.out.println("Values: "+map.values());
12. //Fetching key-value pair
13. System.out.println("Key-Value pairs: "+map.entrySet());
14. }
15. }

Keys: [100, 101, 102]

Values: [Amit, Vijay, Rahul]

Key-Value pairs: [100=Amit, 101=Vijay, 102=Rahul]

### **Java LinkedHashMap Example:remove()**

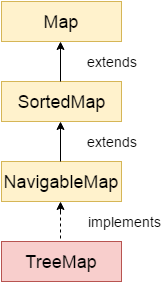
1. **import** java.util.\*;
2. **public** **class** LinkedHashMap3 {
3. **public** **static** **void** main(String args[]) {
4. Map<Integer,String> map=**new** LinkedHashMap<Integer,String>();
5. map.put(101,"Amit");
6. map.put(102,"Vijay");
7. map.put(103,"Rahul");
8. System.out.println("Before invoking remove() method: "+map);
9. map.remove(102);
10. System.out.println("After invoking remove() method: "+map);
11. }
12. }

Output:

Before invoking remove() method: {101=Amit, 102=Vijay, 103=Rahul}

After invoking remove() method: {101=Amit, 103=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.

### **TreeMap class declaration**

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

1. **public** **class** TreeMap<K,V> **extends** AbstractMap<K,V> **implements** NavigableMap<K,V>, Cloneable, Serializable

### **TreeMap class Parameters**

Let's see the Parameters for java.util.TreeMap class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Constructors of Java TreeMap class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| TreeMap() | It is used to construct an empty tree map that will be sorted using the natural  order of its key. |
| TreeMap(Comparator<? super K> comparator) | It is used to construct an empty tree-based map that will be sorted using the  comparator comp. |
| TreeMap(Map<? extends K,? extends V> m) | It is used to initialize a treemap with the entries from **m**, which will be sorted  using the natural order of the keys. |
| TreeMap(SortedMap<K,? extends V> m) | It is used to initialize a treemap with the entries from the SortedMap **sm**, which  will be sorted in the same order as **sm.** |

### **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 TreeMap Example: remove()**

1. **import** java.util.\*;
2. **public** **class** TreeMap2 {
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");
9. System.out.println("Before invoking remove() method");
10. **for**(Map.Entry m:map.entrySet())
11. {
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. map.remove(102);
15. System.out.println("After invoking remove() method");
16. **for**(Map.Entry m:map.entrySet())
17. {
18. System.out.println(m.getKey()+" "+m.getValue());
19. }
20. }
21. }

Output:

Before invoking remove() method

100 Amit

101 Vijay

102 Ravi

103 Rahul

After invoking remove() method

100 Amit

101 Vijay

103 Rahul

### **Java TreeMap Example: NavigableMap**

1. **import** java.util.\*;
2. **class** TreeMap3{
3. **public** **static** **void** main(String args[]){
4. NavigableMap<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");
9. //Maintains descending order
10. System.out.println("descendingMap: "+map.descendingMap());
11. //Returns key-value pairs whose keys are less than or equal to the specified key.
12. System.out.println("headMap: "+map.headMap(102,**true**));
13. //Returns key-value pairs whose keys are greater than or equal to the specified key.
14. System.out.println("tailMap: "+map.tailMap(102,**true**));
15. //Returns key-value pairs exists in between the specified key.
16. System.out.println("subMap: "+map.subMap(100, **false**, 102, **true**));
17. }
18. }

descendingMap: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

headMap: {100=Amit, 101=Vijay, 102=Ravi}

tailMap: {102=Ravi, 103=Rahul}

subMap: {101=Vijay, 102=Ravi}

### **Java TreeMap Example: SortedMap**

1. **import** java.util.\*;
2. **class** TreeMap4{
3. **public** **static** **void** main(String args[]){
4. SortedMap<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");
9. //Returns key-value pairs whose keys are less than the specified key.
10. System.out.println("headMap: "+map.headMap(102));
11. //Returns key-value pairs whose keys are greater than or equal to the specified key.
12. System.out.println("tailMap: "+map.tailMap(102));
13. //Returns key-value pairs exists in between the specified key.
14. System.out.println("subMap: "+map.subMap(100, 102));
15. }
16. }

headMap: {100=Amit, 101=Vijay}

tailMap: {102=Ravi, 103=Rahul}

subMap: {100=Amit, 101=Vijay}

### **What is difference between HashMap and TreeMap?**

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| 1) HashMap can contain one null key. | TreeMap cannot contain any null key. |
| 2) HashMap maintains no order. | TreeMap maintains ascending order. |

### **Java TreeMap Example: Book**

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** MapExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** TreeMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(2,b2);
24. map.put(1,b1);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

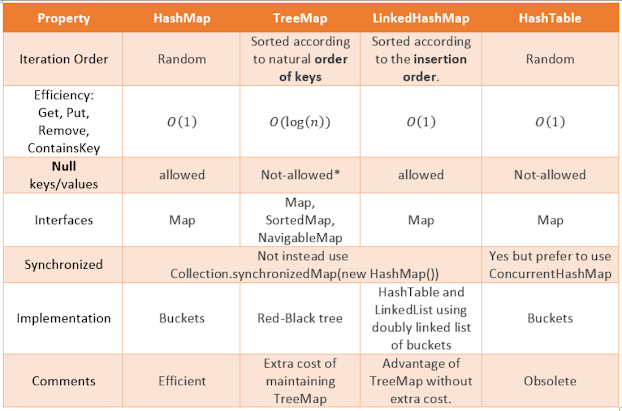
103 Operating System Galvin Wiley 6

## Difference between HashMap, LinkedHashMap and TreeMap in Java

Here are some important difference between TreeMap, LinkedHashMap, and HashMap in Java on point format  
  
**1. Order of Elements**  
HashMap doesn't maintain any order, TreeMap keeps all elements in sorted order, specified by Comparator or object's natural order defined by Comparable. LinkedHashMap keeps elements in the same order they are inserted into map.   
  
**2. Performance**

HashMap gives best performance because there is no overhead, TreeMap gives slower performance because every time you add or remove mapping , it need to sort the whole map. LinkedHashMap gives performance in between,   
  
  
**3. Null keys and values**

HashMap doesn't all null keys but allows null value, but TreeMap doesn't allow null key. LinkedHashMap allows null key.

[](https://javarevisited.blogspot.com/2011/11/collection-interview-questions-answers.html)

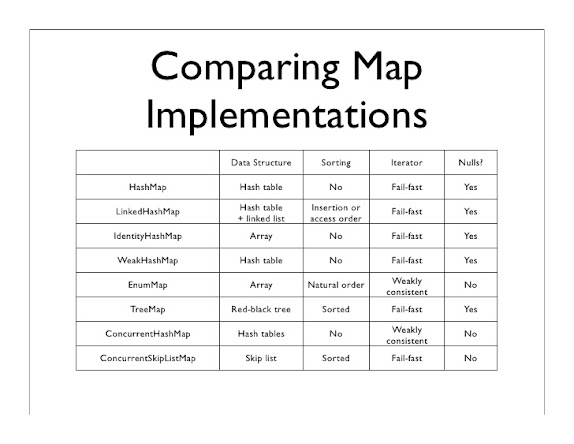
**4. Fail-fast behavior**

Iterator of all map are fail-fast in nature.

**5. Internal implementation**

HashMap is internally based upon hash table data structure, [TreeMap](http://javarevisited.blogspot.sg/2011/12/treemap-java-tutorial-example-program.html)is based upon Red Black Tree and LinkedHashMap uses doubly linked list to keep elements in the same order they are inserted.   
  
  
**6. Synchronization**

None of these map are synchornized.   
  
  
**7. Usage**  
LinkedHashMap also provides a great starting point for creating a Cache object by overriding the **removeEldestEntry()** method. This lets you create a Cache object that can expire data using some criteria that you define.



# 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.

### **Hashtable class declaration**

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

1. **public** **class** Hashtable<K,V> **extends** Dictionary<K,V> **implements** Map<K,V>, Cloneable, Serializable

### **Hashtable class Parameters**

Let's see the Parameters for java.util.Hashtable class.

* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

### **Constructors of Java Hashtable class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| Hashtable() | It creates an empty hashtable having the initial default capacity and load  factor. |
| Hashtable(int capacity) | It accepts an integer parameter and creates a hash table that contains a  specified initial capacity. |
| Hashtable(int capacity, float loadFactor) | It is used to create a hash table having the specified initial capacity and  loadFactor. |
| Hashtable(Map<? extends K,? extends V> t) | It creates a new hash table with the same mappings as the given Map. |

### **Methods of Java Hashtable class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void clear() | It is used to reset the hash table. |
| Object clone() | It returns a shallow copy of the Hashtable. |
| 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. |
| Enumeration elements() | It returns an enumeration of the values in the hash table. |
| Set<Map.Entry<K,V>> entrySet() | It returns a set view of the mappings contained in the map. |
| 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 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 |
| Enumeration<K> keys() | It returns an enumeration of the keys in the hashtable. |
| Set<K> keySet() | It returns a Set view of the keys contained in the map. |
| 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 put(K key, V value) | It inserts the specified value with the specified key in the hash table. |
| void putAll(Map<? extends K,? extends V> t)) | It is used to copy all the key-value pair from map to hashtable. |
| V putIfAbsent(K key, V value) | If the specified key is not already associated with a value (or is mapped to  null) associates it with the given value and returns null, else returns the current  value. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the  hashtable. |
| 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. |
| String toString() | It returns a string representation of the Hashtable object. |
| Collection values() | It returns a collection view of the values contained in the map. |
| boolean contains(Object value) | This method returns true if some value equal to the value exists within the  hash table, else return false. |
| boolean containsValue(Object value) | This method returns true if some value equal to the value exists within the  hash table, else return false. |
| boolean containsKey(Object key) | This method return true if some key equal to the key exists within the hash  table, else return false. |
| boolean isEmpty() | This method returns true if the hash table is empty; returns false if it contains  at least one key. |
| protected void rehash() | It is used to increase the size of the hash table and rehashes all of its keys. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V remove(Object key) | It is used to remove the key and its value. This method returns the value  associated with the key. |
| int size() | This method returns the number of entries in the hash table. |

## Java Hashtable Class

The Hashtable class in Java is a concrete implementation of a Dictionary and was originally a part of java.util package. The Hashtable class creates a hash table by mapping keys to values.

In a hashtable, any non-null object can be used as a key or as a value. The objects used as keys must implement the hashCode and equals methods in order to effectively store and retrieve items from a hashtable.

Hashtable is not just a data structure, but also a **Java Collection API class**. Despite the fact that both the array and hashtable data structures are intended for fast search, i.e. constant time search operation, also known as O(1) search, the fundamental difference between them is that the array requires an index, whereas the hash table requires a key, which could be another object.

This code example generates a hashtable of numbers. It employs the bird numbers as keys:

import java.util.Hashtable;

import java.util.Enumeration;

public class Main {

public static void main(String[] args) {

Enumeration birds;

String key;

*// Creating a Hashtable*

Hashtable<String, String> hashtable =

new Hashtable<String, String>();

*// Adding Key and Value pairs to Hashtable*

hashtable.put("Bird1","Pigeon");

hashtable.put("Bird2","BlueBird");

hashtable.put("Bird3","Swan");

hashtable.put("Bird4","Parrot");

hashtable.put("Bird5","Sparrow");

birds = hashtable.keys();

while(birds.hasMoreElements()) {

key = (String) birds.nextElement();

System.out.println("Key: " +key+ " & Value: " +

hashtable.get(key));

}

}

}

**Output:**

Key: Bird4 & Value: Parrot

Key: Bird3 & Value: Swan

Key: Bird2 & Value: BlueBird

Key: Bird1 & Value: Pigeon

Key: Bird5 & Value: Sparrow

As depicted above, a hashtable does not guarantee the order of records inserted in it.

### **Java Hashtable Example**

1. **import** java.util.\*;
2. **class** Hashtable1{
3. **public** **static** **void** main(String args[]){
4. Hashtable<Integer,String> hm=**new** Hashtable<Integer,String>();
6. hm.put(100,"Amit");
7. hm.put(102,"Ravi");
8. hm.put(101,"Vijay");
9. hm.put(103,"Rahul");
11. **for**(Map.Entry m:hm.entrySet()){
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. }
15. }

Output:

69.5M

1.2K

Difference between JDK, JRE, and JVM

103 Rahul

102 Ravi

101 Vijay

100 Amit

### **Java Hashtable Example: remove()**

1. **import** java.util.\*;
2. **public** **class** Hashtable2 {
3. **public** **static** **void** main(String args[]) {
4. Hashtable<Integer,String> map=**new** Hashtable<Integer,String>();
5. map.put(100,"Amit");
6. map.put(102,"Ravi");
7. map.put(101,"Vijay");
8. map.put(103,"Rahul");
9. System.out.println("Before remove: "+ map);
10. // Remove value for key 102
11. map.remove(102);
12. System.out.println("After remove: "+ map);
13. }
14. }

Output:

Before remove: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

After remove: {103=Rahul, 101=Vijay, 100=Amit}

### **Java Hashtable Example: getOrDefault()**

1. **import** java.util.\*;
2. **class** Hashtable3{
3. **public** **static** **void** main(String args[]){
4. Hashtable<Integer,String> map=**new** Hashtable<Integer,String>();
5. map.put(100,"Amit");
6. map.put(102,"Ravi");
7. map.put(101,"Vijay");
8. map.put(103,"Rahul");
9. //Here, we specify the if and else statement as arguments of the method
10. System.out.println(map.getOrDefault(101, "Not Found"));
11. System.out.println(map.getOrDefault(105, "Not Found"));
12. }
13. }

Output:

Vijay

Not Found

### **Java Hashtable Example: putIfAbsent()**

1. **import** java.util.\*;
2. **class** Hashtable4{
3. **public** **static** **void** main(String args[]){
4. Hashtable<Integer,String> map=**new** Hashtable<Integer,String>();
5. map.put(100,"Amit");
6. map.put(102,"Ravi");
7. map.put(101,"Vijay");
8. map.put(103,"Rahul");
9. System.out.println("Initial Map: "+map);
10. //Inserts, as the specified pair is unique
11. map.putIfAbsent(104,"Gaurav");
12. System.out.println("Updated Map: "+map);
13. //Returns the current value, as the specified pair already exist
14. map.putIfAbsent(101,"Vijay");
15. System.out.println("Updated Map: "+map);
16. }
17. }

Output:

Initial Map: {103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

Updated Map: {104=Gaurav, 103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

Updated Map: {104=Gaurav, 103=Rahul, 102=Ravi, 101=Vijay, 100=Amit}

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.

But there are many differences between HashMap and Hashtable classes that are given below.

|  |  |
| --- | --- |
| **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 an Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

In Iterator, we can read and remove element while traversing element in the collections. Using Enumeration, we can only read element during traversing element in the collections. 2. It can be used with any class of the collection framework.

# Java Collections synchronizedMap() Method

The **synchronizedMap()** method of Java Collections class is used to get a synchronized (thread-safe) map backed by the specified map.

## **Syntax**

Following is the declaration of synchronizedMap() method:

1. **public** **static** <K,V> Map<K,V> synchronizedMap(Map<K,V> m)

## **Parameter**

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Description** | **Required/Optional** |
| m | It is the map which will be wrapped in a synchronized map. | Required |

## **Returns**

The **synchronizedMap()** method returns a synchronized view of the specified Map.

## **Example 1**

1. **import** java.util.\*;
2. **public** **class** CollectionsSynchronizedMapExample1 {
3. **public** **static** **void** main(String[] args) {
4. Map<String, String> map = **new** HashMap<String, String>();
5. map.put("1", "Rahul");
6. map.put("4", "Karan");
7. map.put("3", "Mohan");
8. Map<String, String> synmap = Collections.synchronizedMap(map);
9. System.out.println("Synchronized map is :" + synmap);
10. }
11. }

**Output:**

Synchronized map is :{1=Rahul, 3=Mohan, 4=Karan}

## **Example 2**

1. **import** java.util.\*;
2. **public** **class** CollectionsSynchronizedMapExample2 {
3. **public** **static** **void** main(String[] args) {
4. Map<Integer, Integer> map = **new** HashMap<>();
5. map.put(1, 1001);
6. map.put(2, 1002);
7. map.put(3, 1003);
8. map.put(4, 1004);
9. map.put(5, 1005);
10. System.out.println("Map before Synchronized map: " + map);
11. Map<Integer, Integer> synmap = Collections.synchronizedMap(map);
12. map.remove(4, 1004);
13. System.out.println("Synchronized map after remove(4, 1004):" + synmap);
14. }
15. }

**Output:**

Map before Synchronized map: {1=1001, 2=1002, 3=1003, 4=1004, 5=1005}

Synchronized map after remove(4, 1004):{1=1001, 2=1002, 3=1003, 5=1005}

# Lock in Java

In Java, **Lock** is an interface available in the **Java.util.concurrent.locks** package. Java lock acts as thread synchronization mechanisms that are similar to the synchronized blocks. After some time, a new locking mechanism was introduced. It is very flexible and provides more options in comparison to the Synchronized block.

These are some of the following differences between a Synchronized block and a Lock:

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Factor** | **Synchronized block** | **Lock** |
| 1. | Sequence guarantee | It doesn't provide any guarantee of sequence in which the waiting thread will be access. | Unlike Synchronized block, the Lock interface  handles it. |
| 2. | No timeout | It doesn't have any options of time when the lock is not granted. | The Lock interface provides such options at the  time of granting the lock. |
| 3. | Single method | It can be contained within a single method. | The lock() and unlock() methods of the interface  can be called in different methods. |

In Java, the Lock interface basically provides six methods which are as follows:

### **The lock() method**

The **lock()** method is one of the most important methods of the **Lock** interface. It is used for acquiring the lock. For thread scheduling purposes, the current thread becomes disabled when the lock is not available. The lock() method is a public method that returns void.

#### UnLock():

UnLock() releases the lock on [Object](https://crunchify.com/create-simple-pojo-and-multiple-java-reflection-examples/).

# Java ConcurrentHashMap class

A hash table supporting full concurrency of retrievals and high expected concurrency for updates. This class obeys the same functional specification as Hashtable and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.

| **Sr. No.** | **Key** | **HashTable** | **ConcurrentHashMap** |
| --- | --- | --- | --- |
| 1 | Basic | HashTable is a thread-safe legacy class introduced in the Jdk1.1 | ConcurrentHashmap is a class that was introduced in  jdk1.5 |
| 2 | Locking | It applies lock on the entire collection | ConcurrentHashMap apply locks only at bucket level  called fragment  while adding or updating the map |
| 3 | Performance | It is slower than  ConcurrentHashMap | It is better than HashTable |
| 4. | Null | It doesn't allow null key and value | It allows null key and value |

## **Java ConcurrentHashMap class declaration**

1. **public** **class** ConcurrentHashMap<K,V>
2. **extends** AbstractMap<K,V>
3. **implements** ConcurrentMap<K,V>, Serializable

## **Java ConcurrentHashMap class Example: computeIfAbsent()**

1. //import statement
3. **import** java.util.concurrent.\*;
5. **import** java.util.\*;
7. **public** **class** ConcurrentHashMapcomputeIfAbsentExample1\_1 {
9. **public** **static** **void** main(String[] args)
10. {
11. // crete a HashMap and add some values
12. HashMap<String, Integer> mapcon   = **new** HashMap<>();
13. mapcon.put("k1", 100);
14. mapcon.put("k2", 200);
15. mapcon.put("k3", 300);
16. mapcon.put("k4", 400);
17. System.out.println("HashMap values :\n " + mapcon.toString());
18. mapcon.computeIfAbsent("k5", k -> 200 + 300);
19. mapcon.computeIfAbsent("k6", k -> 60 \* 10);
20. System.out.println("New HashMap after computeIfAbsent :\n "+ mapcon);
21. }
22. }

**Output:**

HashMap values :

{k1=100, k2=200, k3=300, k4=400}

New HashMap after computeIfAbsent :

{k1=100, k2=200, k3=300, k4=400, k5=500, k6=600}

## **Java ConcurrentHashMap Class Example: containsValue()**

1. **import** java.util.\*;
2. **import** java.util.concurrent.\*;
4. **public** **class** ConcurrentHashMapcontainsValueExample1\_1  {
5. **public** **static** **void** main(String[] args)
6. {
7. ConcurrentHashMap<String, Integer>  mymap = **new** ConcurrentHashMap<String,  Integer>();
9. mymap.put("AAA", 10);
10. mymap.put("BBB", 15);
11. mymap.put("CCC", 25);
12. mymap.put("DDD", 255);
13. mymap.put("EEE",30);
14. System.out.println(" Mappings are: " +mymap);
16. System.out.println("is 255  present? ::  "
17. + mymap.containsValue(255));
19. }
20. }

**Output:**

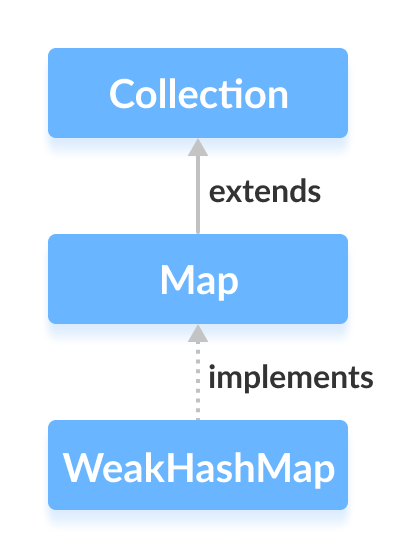
Mappings are: {AAA=10, CCC=25, BBB=15, EEE=30, DDD=255}

is 255 present? :: true

# Java WeakHashMap

The WeakHashMap class of the Java collections framework provides the feature of the hash table data structure..

It implements the [Map interface](https://www.programiz.com/java-programming/map).



**Note**: Keys of the weak hashmap are of the **WeakReference** type.

The object of a weak reference type can be garbage collected in Java if the reference is no longer used in the program.

Let us learn to create a weak hash map first. Then, we will learn how it differs from a hashmap.

## Create a WeakHashMap

In order to create a weak hashmap, we must import the java.util.WeakHashMap package first. Once we import the package, here is how we can create weak hashmaps in Java.

//WeakHashMap creation with capacity 8 and load factor 0.6

WeakHashMap<Key, Value> numbers = new WeakHashMap<>(8, 0.6);

In the above code, we have created a weak hashmap named numbers.

Here,

* Key - a unique identifier used to associate each element (value) in a map
* Value - elements associated by keys in a map

Notice the part new WeakHashMap<>(8, 0.6). Here, the first parameter is **capacity** and the second parameter is **loadFactor**.

* **capacity** - The capacity of this map is 8. Meaning, it can store 8 entries.
* **loadFactor** - The load factor of this map is 0.6. This means whenever our hash table is filled by 60%, the entries are moved to a new hash table of double the size of the original hash table.

**Default capacity and load factor**

It is possible to create a weak hashmap without defining its capacity and load factor. For example,

// WeakHashMap with default capacity and load factor

WeakHashMap<Key, Value> numbers1 = new WeakHashMap<>();

By default,

* the capacity of the map will be 16
* the load factor will be 0.75

## Differences Between HashMap and WeakHashMap

Let us see the implementation of a weak hashmap in Java.

import java.util.WeakHashMap;

class Main {

public static void main(String[] args) {

// Creating WeakHashMap of numbers

WeakHashMap<String, Integer> numbers = new WeakHashMap<>();

String two = new String("Two");

Integer twoValue = 2;

String four = new String("Four");

Integer fourValue = 4;

// Inserting elements

numbers.put(two, twoValue);

numbers.put(four, fourValue);

System.out.println("WeakHashMap: " + numbers);

// Make the reference null

two = null;

// Perform garbage collection

System.gc();

System.out.println("WeakHashMap after garbage collection: " + numbers);

}

}

**Output**

WeakHashMap: {Four=4, Two=2}

WeakHashMap after garbage collection: {Four=4}

As we can see, when the key two of a weak hashmap is set to null and perform garbage collection, the key is removed.

It is because unlike hashmaps, keys of weak hashmaps are of **weak reference** type. This means the entry of a map are removed by the garbage collector if the key to that entry is no longer used. This is useful to save resources.

Now let us see the same implementation in a hashmap.

import java.util.HashMap;

class Main {

public static void main(String[] args) {

// Creating HashMap of even numbers

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

String two = new String("Two");

Integer twoValue = 2;

String four = new String("Four");

Integer fourValue = 4;

// Inserting elements

numbers.put(two, twoValue);

numbers.put(four, fourValue);

System.out.println("HashMap: " + numbers);

// Make the reference null

two = null;

// Perform garbage collection

System.gc();

System.out.println("HashMap after garbage collection: " + numbers);

}

}

**Output**

HashMap: {Four=4, Two=2}

HashMap after garbage collection: {Four=4, Two=2}

Here, when the key two of the hashmap is set to null and perform garbage collection, the key is not removed.

This is because unlike weak hashmaps keys of hashmaps are of **strong reference** type. This means the entry of a map is not removed by the garbage collector even though the key to that entry is no longer used.

**Note**: All functionalities of hashmaps and weak hashmaps are similar except keys of a weak hashmap are of weak reference, whereas keys of a hashmap are of strong reference.

## Creating WeakHashMap from Other Maps

Here is how we can create a weak hashmap from other maps.

import java.util.HashMap;

import java.util.WeakHashMap;

class Main {

public static void main(String[] args) {

// Creating a hashmap of even numbers

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

String two = new String("Two");

Integer twoValue = 2;

evenNumbers.put(two, twoValue);

System.out.println("HashMap: " + evenNumbers);

// Creating a weak hash map from other hashmap

WeakHashMap<String, Integer> numbers = new WeakHashMap<>(evenNumbers);

System.out.println("WeakHashMap: " + numbers);

}

}

**Output**

HashMap: {Two=2}

WeakHashMap: {Two=2}

# Difference between Comparable and Comparator

Comparable and Comparator both are interfaces and can be used to sort collection elements.

However, there are many differences between Comparable and Comparator interfaces that are given below.

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price. | The Comparator provides **multiple sorting sequences**. In other  words, we can sort the collection on the basis of multiple  elements such as id, name, and price etc. |
| 2) Comparable **affects the original class**, i.e., the actual class is modified. | Comparator **doesn't affect the original class**, i.e., the actual  class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is present in **java.lang** package. | A Comparator is present in the **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by  **Collections.sort(List, Comparator)** method. |

## **Java Comparable Example**

Let's see the example of a Comparable interface that sorts the list elements on the basis of age.

*File: TestSort3.java*

//Java Program to demonstrate the use of Java Comparable.

1. //Creating a class which implements Comparable Interface
2. **import** java.util.\*;
3. **import** java.io.\*;
4. **class** Student **implements** Comparable<Student>{
5. **int** rollno;
6. String name;
7. **int** age;
8. Student(**int** rollno,String name,**int** age){
9. **this**.rollno=rollno;
10. **this**.name=name;
11. **this**.age=age;
12. }
13. **public** **int** compareTo(Student st){
14. **if**(age==st.age)
15. **return** 0;
16. **else** **if**(age>st.age)
17. **return** 1;
18. **else**
19. **return** -1;
20. }
21. }
22. //Creating a test class to sort the elements
23. **public** **class** TestSort3{
24. **public** **static** **void** main(String args[]){
25. ArrayList<Student> al=**new** ArrayList<Student>();
26. al.add(**new** Student(101,"Vijay",23));
27. al.add(**new** Student(106,"Ajay",27));
28. al.add(**new** Student(105,"Jai",21));
30. Collections.sort(al);
31. **for**(Student st:al){
32. System.out.println(st.rollno+" "+st.name+" "+st.age);
33. }
34. }
35. }

Output:

105 Jai 21

101 Vijay 23

106 Ajay 27

## **Java Comparator Example**

Let's see an example of the Java Comparator interface where we are sorting the elements of a list using different comparators.

**Student.java**

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }

**AgeComparator.java**

1. **import** java.util.\*;
2. **class** AgeComparator **implements** Comparator<Student>{
3. **public** **int** compare(Student s1,Student s2){
4. **if**(s1.age==s2.age)
5. **return** 0;
6. **else** **if**(s1.age>s2.age)
7. **return** 1;
8. **else**
9. **return** -1;
10. }
11. }

**NameComparator.java**

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

1. **import** java.util.\*;
2. **class** NameComparator **implements** Comparator<Student>{
3. **public** **int** compare(Student s1,Student s2){
4. **return** s1.name.compareTo(s2.name);
5. }
6. }

**TestComparator.java**

In this class, we are printing the values of the object by sorting on the basis of name and age.

1. //Java Program to demonstrate the use of Java Comparator
2. **import** java.util.\*;
3. **import** java.io.\*;
4. **class** TestComparator{
5. **public** **static** **void** main(String args[]){
6. //Creating a list of students
7. ArrayList<Student> al=**new** ArrayList<Student>();
8. al.add(**new** Student(101,"Vijay",23));
9. al.add(**new** Student(106,"Ajay",27));
10. al.add(**new** Student(105,"Jai",21));
12. System.out.println("Sorting by Name");
13. //Using NameComparator to sort the elements
14. Collections.sort(al,**new** NameComparator());
15. //Traversing the elements of list
16. **for**(Student st: al){
17. System.out.println(st.rollno+" "+st.name+" "+st.age);
18. }
20. System.out.println("sorting by Age");
21. //Using AgeComparator to sort the elements
22. Collections.sort(al,**new** AgeComparator());
23. //Travering the list again
24. **for**(Student st: al){
25. System.out.println(st.rollno+" "+st.name+" "+st.age);
26. }
28. }
29. }

Output:

Sorting by Name

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by Age

105 Jai 21

101 Vijay 23

106 Ajay 27

1. Student.java
2. AgeComparator.java
3. NameComparator.java
4. Simple.java

**Student.java**

This class contains three fields rollno, name and age and a parameterized constructor.

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }

**AgeComparator.java**

This class defines comparison logic based on the age. If the age of the first object is greater than the second, we are returning a positive value. It can be anyone such as 1, 2, 10. If the age of the first object is less than the second object, we are returning a negative value, it can be any negative value, and if the age of both objects is equal, we are returning 0.

1. **import** java.util.\*;
2. **class** AgeComparator **implements** Comparator{
3. **public** **int** compare(Object o1,Object o2){
4. Student s1=(Student)o1;
5. Student s2=(Student)o2;
7. **if**(s1.age==s2.age)
8. **return** 0;
9. **else** **if**(s1.age>s2.age)
10. **return** 1;
11. **else**
12. **return** -1;
13. }
14. }

**NameComparator.java**

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

1. **import** java.util.\*;
2. **class** NameComparator **implements** Comparator{
3. **public** **int** compare(Object o1,Object o2){
4. Student s1=(Student)o1;
5. Student s2=(Student)o2;
7. **return** s1.name.compareTo(s2.name);
8. }
9. }

**Simple.java**

In this class, we are printing the values of the object by sorting on the basis of name and age.

1. **import** java.util.\*;
2. **import** java.io.\*;
4. **class** Simple{
5. **public** **static** **void** main(String args[]){
7. ArrayList al=**new** ArrayList();
8. al.add(**new** Student(101,"Vijay",23));
9. al.add(**new** Student(106,"Ajay",27));
10. al.add(**new** Student(105,"Jai",21));
12. System.out.println("Sorting by Name");
14. Collections.sort(al,**new** NameComparator());
15. Iterator itr=al.iterator();
16. **while**(itr.hasNext()){
17. Student st=(Student)itr.next();
18. System.out.println(st.rollno+" "+st.name+" "+st.age);
19. }
21. System.out.println("Sorting by age");
23. Collections.sort(al,**new** AgeComparator());
24. Iterator itr2=al.iterator();
25. **while**(itr2.hasNext()){
26. Student st=(Student)itr2.next();
27. System.out.println(st.rollno+" "+st.name+" "+st.age);
28. }

31. }
32. }

Sorting by Name

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by age

105 Jai 21

101 Vijay 23

106 Ajay 27