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.

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 classes, i.e.,
2. Algorithm-Search,sort

### **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.



Map –Has Map internal working /has set internal working

## **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.,

Iterator<T> iterator()

Public Boolean hasNext(){

}

## **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.

## **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.

List<String> l=new Arraylist();

l.add(“Srinu”);

## **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   Iterator itr = list.iterator();
10. Iterator itr=list.iterator();   //looping the elements
11. **while**(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }
16. Ravi

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

## **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

## **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 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. }
12. }

### **Iterating ArrayList using Iterator**

Let's see an example to traverse ArrayList elements using the Iterator interface.

**FileName:** ArrayListExample2.java

1. **import** java.util.\*;
2. **public** **class** ArrayListExample2{
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 Iterator
10. Iterator itr=list.iterator();//getting the Iterator
11. **while**(itr.hasNext()){//check if iterator has the elements
12. System.out.println(itr.next());//printing the element and move to next
13. }
14. }
15. }

### **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. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=ArrayListExample4)

**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

# **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 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 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.

### **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. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=HashMapExample1" \t "_blank)**

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. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=HashMapExample2" \t "_blank)**

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: 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** HashMap<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(1,b1);
24. map.put(2,b2);
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. }

**[Test it Now](https://www.javatpoint.com/opr/test.jsp?filename=MapExample" \t "_blank)**

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications and Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

1. Vijay
2. Ravi
3. Ajay

# **[Java Queue Interface](https://www.javatpoint.com/collections-in-java)**

[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.](https://www.javatpoint.com/collections-in-java)

[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.](https://www.javatpoint.com/collections-in-java)

## **[Features of a Queue](https://www.javatpoint.com/collections-in-java)**

[The following are some important features of a queue.](https://www.javatpoint.com/collections-in-java)

* [As discussed earlier, FIFO concept is used for insertion and deletion of elements from a queue.](https://www.javatpoint.com/collections-in-java)
* [The Java Queue provides support for all of the methods of the Collection interface including deletion, insertion, etc.](https://www.javatpoint.com/collections-in-java)
* [PriorityQueue, ArrayBlockingQueue and LinkedList are the implementations that are used most frequently.](https://www.javatpoint.com/collections-in-java)
* [The NullPointerException is raised, if any null operation is done on the BlockingQueues.](https://www.javatpoint.com/collections-in-java)
* [Those Queues that are present in the util package are known as Unbounded Queues.](https://www.javatpoint.com/collections-in-java)
* [Those Queues that are present in the util.concurrent package are known as bounded Queues.](https://www.javatpoint.com/collections-in-java)
* [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.](https://www.javatpoint.com/collections-in-java)

## **[PriorityQueue Class](https://www.javatpoint.com/collections-in-java)**

[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.](https://www.javatpoint.com/collections-in-java)

### **[PriorityQueue Class Declaration](https://www.javatpoint.com/collections-in-java)**

[Let's see the declaration for java.util.PriorityQueue class.](https://www.javatpoint.com/collections-in-java)

1. **[public](https://www.javatpoint.com/collections-in-java)****[class](https://www.javatpoint.com/collections-in-java)**[PriorityQueue<E>](https://www.javatpoint.com/collections-in-java)**[extends](https://www.javatpoint.com/collections-in-java)**[AbstractQueue<E>](https://www.javatpoint.com/collections-in-java)**[implements](https://www.javatpoint.com/collections-in-java)**[Serializable](https://www.javatpoint.com/collections-in-java)

### **[Java PriorityQueue Example](https://www.javatpoint.com/collections-in-java)**

**[FileName:](https://www.javatpoint.com/collections-in-java)**[TestCollection12.java](https://www.javatpoint.com/collections-in-java)

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

[**[Test it Now](https://www.javatpoint.com/collections-in-java)**](https://www.javatpoint.com/opr/test.jsp?filename=TestCollection12)

**[Output:](https://www.javatpoint.com/collections-in-java)**

[head:Amit](https://www.javatpoint.com/collections-in-java)

[head:Amit](https://www.javatpoint.com/collections-in-java)

[iterating the queue elements:](https://www.javatpoint.com/collections-in-java)

[Amit](https://www.javatpoint.com/collections-in-java)

[Jai](https://www.javatpoint.com/collections-in-java)

[Karan](https://www.javatpoint.com/collections-in-java)

[Vijay](https://www.javatpoint.com/collections-in-java)

[Rahul](https://www.javatpoint.com/collections-in-java)

[after removing two elements:](https://www.javatpoint.com/collections-in-java)

[Karan](https://www.javatpoint.com/collections-in-java)

[Rahul](https://www.javatpoint.com/collections-in-java)

[Vijay](https://www.javatpoint.com/collections-in-java)

# **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".**

## **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.

## **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

collection vs collections

framework means interface and class

list – array list and linked list and set

Iterstor methods – Integers and strings