Java Collection FrameWork:

#### What is Collection in Java

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

#### 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

### 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.



### Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

|  |  |  |
| --- | --- | --- |
| **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. |

## 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.

The classes that implement the List interface are given below.

## 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:

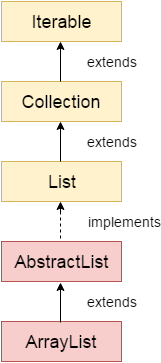
Ravi

Vijay

Ravi

Ajay

# 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 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 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 array works at the index basis.
* In ArrayList, manipulation is 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.

### Hierarchy of ArrayList class

As shown in the above diagram, Java ArrayList class extends AbstractList class which implements 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 specified comparator. |
| Spliterator<E> spliterator() | It is used to create spliterator over the elements in a list. |
| List<E> subList(int fromIndex, int toIndex) | It is used to fetch all the elements 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 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 objects in it. If you try to add another type of object, it gives *compile time error*.

For more information on Java generics, click here [Java Generics Tutorial](https://www.javatpoint.com/generics-in-java).

### Java ArrayList Example

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. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=ArrayListExample1)

**Output:**

[Mango, Apple, Banana, Grapes]

### Iterating ArrayList using Iterator

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

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. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=ArrayListExample2)

**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

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:**

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=ArrayListExample3)

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.

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.

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.

### Iterating Collection through remaining ways

Let's see an example to traverse the ArrayList elements through other ways

1. **import** java.util.\*;
2. **class** ArrayList4{
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");
10. System.out.println("Traversing list through List Iterator:");
11. //Here, element iterates in reverse order
12. ListIterator<String> list1=list.listIterator(list.size());
13. **while**(list1.hasPrevious())
14. {
15. String str=list1.previous();
16. System.out.println(str);
17. }
18. System.out.println("Traversing list through for loop:");
19. **for**(**int** i=0;i<list.size();i++)
20. {
21. System.out.println(list.get(i));
22. }
24. System.out.println("Traversing list through forEach() method:");
25. //The forEach() method is a new feature, introduced in Java 8.
26. list.forEach(a->{ //Here, we are using lambda expression
27. System.out.println(a);
28. });
30. System.out.println("Traversing list through forEachRemaining() method:");
31. Iterator<String> itr=list.iterator();
32. itr.forEachRemaining(a-> //Here, we are using lambda expression
33. {
34. System.out.println(a);
35. });
36. }
37. }

**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.

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:**

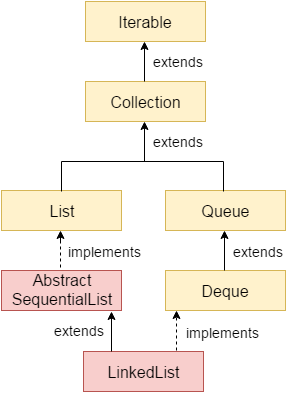
101 Sonoo 23

102 Ravi 21

103 Hanumat 25



Java 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.

29M

673

OOPs Concepts in Java



### 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

Java HashSet

Java HashSet class hierarchy

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.

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) It is used to add the specified element to this set if it is not already present.

2) void clear() It is used to remove all of the elements from the set.

3) object clone() It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.

4) boolean contains(Object o) It is used to return true if this set contains the specified element.

5) boolean isEmpty() It is used to return true if this set contains no elements.

6) Iterator<E> iterator() It is used to return an iterator over the elements in this set.

7) boolean remove(Object o) It is used to remove the specified element from this set if it is present.

8) int size() It is used to return the number of elements in the set.

9) Spliterator<E> spliterator() 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.

import java.util.\*;

class HashSet1{

public static void main(String args[]){

//Creating HashSet and adding elements

HashSet<String> set=new HashSet();

set.add("One");

set.add("Two");

set.add("Three");

set.add("Four");

set.add("Five");

Iterator<String> i=set.iterator();

while(i.hasNext())

{

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

}

}

}

Five

One

Four

Two

Three

Java HashSet example ignoring duplicate elements

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

import java.util.\*;

class HashSet2{

public static void main(String args[]){

//Creating HashSet and adding elements

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

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

//Traversing elements

For(String s:set)

{sys(s);

}

}

}

}

Ajay

Vijay

Ravi

Java HashSet example to remove elements

Here, we see different ways to remove an element.

import java.util.\*;

class HashSet3{

public static void main(String args[]){

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

set.add("Ravi");

set.add("Vijay");

set.add("Arun");

set.add("Sumit");

System.out.println("An initial list of elements: "+set);

//Removing specific element from HashSet

set.remove("Ravi");

System.out.println("After invoking remove(object) method: "+set);

HashSet<String> set1=new HashSet<String>();

set1.add("Ajay");

set1.add("Gaurav");

set.addAll(set1);

System.out.println("Updated List: "+set);

//Removing all the new elements from HashSet

set.removeAll(set1);

System.out.println("After invoking removeAll() method: "+set);

//Removing elements on the basis of specified condition

set.removeIf(str->str.contains("Vijay"));

System.out.println("After invoking removeIf() method: "+set);

//Removing all the elements available in the set

set.clear();

System.out.println("After invoking clear() method: "+set);

}

}

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

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

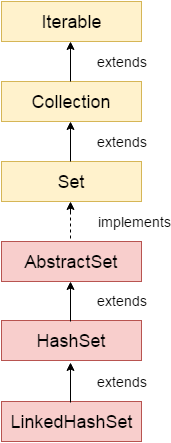
Updated List: [Vijay, Arun, Gaurav, Sumit, Ajay]

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

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

After invoking clear() method: []

# Java LinkedHashSet class



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

The important points about Java LinkedHashSet class are:

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

## Hierarchy of LinkedHashSet class

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

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 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 Java LinkedHashSet class. Here you can notice that the elements iterate in insertion order.

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. }

One

Two

Three

Four

Five

Java LinkedHashSet example ignoring duplicate Elements

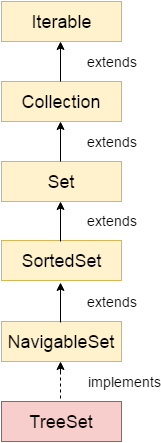
1. **import** java.util.\*;
2. **class** LinkedHashSet2{
3. **public** **static** **void** main(String args[]){
4. LinkedHashSet<String> al=**new** LinkedHashSet<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. Iterator<String> itr=al.iterator();
10. **while**(itr.hasNext()){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

Ravi

Vijay

Ajay

# Java TreeSet class



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

The important points about Java TreeSet class are:

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

### Hierarchy of TreeSet class

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

### TreeSet class declaration

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

31.6M

611

Difference between JDK, JRE, and JVM

1. **public** **class** TreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>, Cloneable, Serializable

### 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 comparator that arranged elements in order. |
| Iterator descendingIterator() | It is used 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 Example 1:

Let's see a simple example of Java TreeSet.

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. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestCollection11)

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. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestCollection11)

Output:

Traversing element through Iterator in descending order

Vijay

Ravi

Ajay

Traversing element through NavigableSet in descending order

Vijay

Ravi

Ajay

[**Next →**](https://www.javatpoint.com/java-hashmap)[**← Prev**](https://www.javatpoint.com/java-priorityqueue)

# 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. |

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 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)

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)

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.

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]

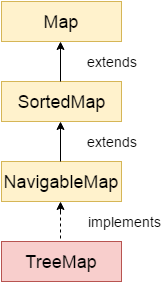
Values: [Amit, Vijay, Rahul]

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. }

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.

31.2M

715

Features of Java - Javatpoint

* **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 Collections class

Java collection class is used exclusively with static methods that operate on or return collections. It inherits Object class.

The important points about Java Collections class are:

* Java Collection class supports the **polymorphic algorithms** that operate on collections.
* Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.

## Collections class declaration

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

1. **public** **class** Collections **extends** Object

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Modifier & Type** | **Methods** | **Descriptions** |
| 1) | static <T> boolean | [addAll()](https://www.javatpoint.com/java-collections-addall-method) | It is used to adds all of the specified elements to the specified collection. |
| 2) | static <T> Queue<T> | [asLifoQueue()](https://www.javatpoint.com/java-collections-aslifoqueue-method) | It returns a view of a Deque as a Last-in-first-out (LIFO) Queue. |
| 3) | static <T> int | [binarySearch()](https://www.javatpoint.com/java-collections-binarysearch-method) | It searches the list for the specified object and returns their position in a sorted list. |
| 4) | static <E> Collection<E> | [checkedCollection()](https://www.javatpoint.com/java-collections-checkedcollection-method) | It is used to returns a dynamically typesafe view of the specified collection. |
| 5) | static <E> List<E> | [checkedList()](https://www.javatpoint.com/java-collections-checkedlist-method) | It is used to returns a dynamically typesafe view of the specified list. |
| 6) | static <K,V> Map<K,V> | [checkedMap()](https://www.javatpoint.com/java-collections-checkedmap-method) | It is used to returns a dynamically typesafe view of the specified map. |
| 7) | static <K,V> NavigableMap<K,V> | [checkedNavigableMap()](https://www.javatpoint.com/java-collections-checkednavigablemap-method) | It is used to returns a dynamically typesafe view of the specified navigable map. |
| 8) | static <E> NavigableSet<E> | [checkedNavigableSet()](https://www.javatpoint.com/java-collections-checkednavigableset-method) | It is used to returns a dynamically typesafe view of the specified navigable set. |
| 9) | static <E> Queue<E> | [checkedQueue()](https://www.javatpoint.com/java-collections-checkedqueue-method) | It is used to returns a dynamically typesafe view of the specified queue. |
| 10) | static <E> Set<E> | [checkedSet()](https://www.javatpoint.com/java-collections-checkedset-method) | It is used to returns a dynamically typesafe view of the specified set. |
| 11) | static <K,V> SortedMap<K,V> | [checkedSortedMap()](https://www.javatpoint.com/java-collections-checkedsortedmap-method) | It is used to returns a dynamically typesafe view of the specified sorted map. |
| 12) | static <E> SortedSet<E> | [checkedSortedSet()](https://www.javatpoint.com/java-collections-checkedsortedset-method) | It is used to returns a dynamically typesafe view of the specified sorted set. |
| 13) | static <T> void | [copy()](https://www.javatpoint.com/java-collections-copy-method) | It is used to copy all the elements from one list into another list. |
| 14) | static boolean | [disjoint()](https://www.javatpoint.com/java-collections-disjoint-method) | It returns true if the two specified collections have no elements in common. |
| 15) | static <T> Enumeration<T> | [emptyEnumeration()](https://www.javatpoint.com/java-collections-emptyenumeration-method) | It is used to get an enumeration that has no elements. |
| 16) | static <T> Iterator<T> | [emptyIterator()](https://www.javatpoint.com/java-collections-emptyiterator-method) | It is used to get an Iterator that has no elements. |
| 17) | static <T> List<T> | [emptyList()](https://www.javatpoint.com/java-collections-emptylist-method) | It is used to get a List that has no elements. |
| 18) | static <T> ListIterator<T> | [emptyListIterator()](https://www.javatpoint.com/java-collections-emptylistiterator-method) | It is used to get a List Iterator that has no elements. |
| 19) | static <K,V> Map<K,V> | [emptyMap()](https://www.javatpoint.com/java-collections-emptymap-method) | It returns an empty map which is immutable. |
| 20) | static <K,V> NavigableMap<K,V> | [emptyNavigableMap()](https://www.javatpoint.com/java-collections-emptynavigablemap-method) | It returns an empty navigable map which is immutable. |
| 21) | static <E> NavigableSet<E> | [emptyNavigableSet()](https://www.javatpoint.com/java-collections-emptynavigableset-method) | It is used to get an empty navigable set which is immutable in nature. |
| 22) | static <T> Set<T> | [emptySet()](https://www.javatpoint.com/java-collections-emptyset-method) | It is used to get the set that has no elements. |
| 23) | static <K,V> SortedMap<K,V> | [emptySortedMap()](https://www.javatpoint.com/java-collections-emptysortedmap-method) | It returns an empty sorted map which is immutable. |
| 24) | static <E> SortedSet<E> | [emptySortedSet()](https://www.javatpoint.com/java-collections-emptysortedset-method) | It is used to get the sorted set that has no elements. |
| 25) | static <T> Enumeration<T> | [enumeration()](https://www.javatpoint.com/java-collections-enumeration-method) | It is used to get the enumeration over the specified collection. |
| 26) | static <T> void | [fill()](https://www.javatpoint.com/java-collections-fill-method) | It is used to replace all of the elements of the specified list with the specified elements. |
| 27) | static int | [frequency()](https://www.javatpoint.com/java-collections-frequency-method) | It is used to get the number of elements in the specified collection equal to the specified object. |
| 28) | static int | [indexOfSubList()](https://www.javatpoint.com/java-collections-indexofsublist-method) | It is used to get the starting position of the first occurrence of the specified target list within the specified source list. It returns -1 if there is no such occurrence in the specified list. |
| 29) | static int | [lastIndexOfSubList()](https://www.javatpoint.com/java-collections-lastindexofsublist-method) | It is used to get the starting position of the last occurrence of the specified target list within the specified source list. It returns -1 if there is no such occurrence in the specified list. |
| 30) | static <T> ArrayList<T> | [list()](https://www.javatpoint.com/java-collections-list-method) | It is used to get an array list containing the elements returned by the specified enumeration in the order in which they are returned by the enumeration. |
| 31) | static <T extends Object & Comparable<? super T>> T | [max()](https://www.javatpoint.com/java-collections-max-method) | It is used to get the maximum value of the given collection, according to the natural ordering of its elements. |
| 32) | static <T extends Object & Comparable<? super T>> T | [min()](https://www.javatpoint.com/java-collections-min-method) | It is used to get the minimum value of the given collection, according to the natural ordering of its elements. |
| 33) | static <T> List<T> | [nCopies()](https://www.javatpoint.com/java-collections-ncopies-method) | It is used to get an immutable list consisting of **n** copies of the specified object. |
| 34) | static <E> Set<E> | [newSetFromMap()](https://www.javatpoint.com/java-collections-newsetfrommap-method) | It is used to return a set backed by the specified map. |
| 35) | static <T> boolean | [replaceAll()](https://www.javatpoint.com/java-collections-replaceall-method) | It is used to replace all occurrences of one specified value in a list with the other specified value. |
| 36) | static void | [reverse()](https://www.javatpoint.com/java-collections-reverse-method) | It is used to reverse the order of the elements in the specified list. |
| 37) | static <T> Comparator<T> | [reverseOrder()](https://www.javatpoint.com/java-collections-reverseorder-method) | It is used to get the comparator that imposes the reverse of the natural ordering on a collection of objects which implement the Comparable interface. |
| 38) | static void | [rotate()](https://www.javatpoint.com/java-collections-rotate-method) | It is used to rotate the elements in the specified list by a given distance. |
| 39) | static void | [shuffle()](https://www.javatpoint.com/java-collections-shuffle-method) | It is used to randomly reorders the specified list elements using a default randomness. |
| 40) | static <T> Set<T> | [singleton()](https://www.javatpoint.com/java-collections-singleton-method) | It is used to get an immutable set which contains only the specified object. |
| 41) | static <T> List<T> | [singletonList()](https://www.javatpoint.com/java-collections-singletonlist-method) | It is used to get an immutable list which contains only the specified object. |
| 42) | static <K,V> Map<K,V> | [singletonMap()](https://www.javatpoint.com/java-collections-singletonmap-method) | It is used to get an immutable map, mapping only the specified key to the specified value. |
| 43) | static <T extends Comparable<? super T>>void | [sort()](https://www.javatpoint.com/java-collections-sort-method) | It is used to sort the elements presents in the specified list of collection in ascending order. |
| 44) | static void | [swap()](https://www.javatpoint.com/java-collections-swap-method) | It is used to swap the elements at the specified positions in the specified list. |
| 45) | static <T> Collection<T> | [synchronizedCollection()](https://www.javatpoint.com/java-collections-synchronizedcollection-method) | It is used to get a synchronized (thread-safe) collection backed by the specified collection. |
| 46) | static <T> List<T> | [synchronizedList()](https://www.javatpoint.com/java-collections-synchronizedlist-method) | It is used to get a synchronized (thread-safe) collection backed by the specified list. |
| 47) | static <K,V> Map<K,V> | [synchronizedMap()](https://www.javatpoint.com/java-collections-synchronizedmap-method) | It is used to get a synchronized (thread-safe) map backed by the specified map. |
| 48) | static <K,V> NavigableMap<K,V> | [synchronizedNavigableMap()](https://www.javatpoint.com/java-collections-synchronizednavigablemap-method) | It is used to get a synchronized (thread-safe) navigable map backed by the specified navigable map. |
| 49) | static <T> NavigableSet<T> | [synchronizedNavigableSet()](https://www.javatpoint.com/java-collections-synchronizednavigableset-method) | It is used to get a synchronized (thread-safe) navigable set backed by the specified navigable set. |
| 50) | static <T> Set<T> | [synchronizedSet()](https://www.javatpoint.com/java-collections-synchronizedset-method) | It is used to get a synchronized (thread-safe) set backed by the specified set. |
| 51) | static <K,V> SortedMap<K,V> | [synchronizedSortedMap()](https://www.javatpoint.com/java-collections-synchronizedsortedmap-method) | It is used to get a synchronized (thread-safe) sorted map backed by the specified sorted map. |
| 52) | static <T> SortedSet<T> | [synchronizedSortedSet()](https://www.javatpoint.com/java-collections-synchronizedsortedset-method) | It is used to get a synchronized (thread-safe) sorted set backed by the specified sorted set. |
| 53) | static <T> Collection<T> | [unmodifiableCollection()](https://www.javatpoint.com/java-collections-unmodifiablecollection-method) | It is used to get an unmodifiable view of the specified collection. |
| 54) | static <T> List<T> | [unmodifiableList()](https://www.javatpoint.com/java-collections-unmodifiablelist-method) | It is used to get an unmodifiable view of the specified list. |
| 55) | static <K,V> Map<K,V> | [unmodifiableMap()](https://www.javatpoint.com/java-collections-unmodifiablemap-method) | It is used to get an unmodifiable view of the specified map. |
| 56) | static <K,V> NavigableMap<K,V> | [unmodifiableNavigableMap()](https://www.javatpoint.com/java-collections-unmodifiablenavigablemap-method) | It is used to get an unmodifiable view of the specified navigable map. |
| 57) | static <T> NavigableSet<T> | [unmodifiableNavigableSet()](https://www.javatpoint.com/java-collections-unmodifiablenavigableset-method) | It is used to get an unmodifiable view of the specified navigable set. |
| 58) | static <T> Set<T> | [unmodifiableSet()](https://www.javatpoint.com/java-collections-unmodifiableset-method) | It is used to get an unmodifiable view of the specified set. |
| 59) | static <K,V> SortedMap<K,V> | [unmodifiableSortedMap()](https://www.javatpoint.com/java-collections-unmodifiablesortedmap-method) | It is used to get an unmodifiable view of the specified sorted map. |
| 60 | static <T> SortedSet<T> | [unmodifiableSortedSet()](https://www.javatpoint.com/java-collections-unmodifiablesortedset-method) | It is used to get an unmodifiable view of the specified sorted set. |

## Java Collections Example

1. **import** java.util.\*;
2. **public** **class** CollectionsExample {
3. **public** **static** **void** main(String a[]){
4. List<String> list = **new** ArrayList<String>();
5. list.add("C");
6. list.add("Core Java");
7. list.add("Advance Java");
8. System.out.println("Initial collection value:"+list);
9. Collections.addAll(list, "Servlet","JSP");
10. System.out.println("After adding elements collection value:"+list);
11. String[] strArr = {"C#", ".Net"};
12. Collections.addAll(list, strArr);
13. System.out.println("After adding array collection value:"+list);
14. }
15. }

Output:

Competitive questions on Structures in Hindi

Keep Watching

Initial collection value:[C, Core Java, Advance Java]

After adding elements collection value:[C, Core Java, Advance Java, Servlet, JSP]

After adding array collection value:[C, Core Java, Advance Java, Servlet, JSP, C#, .Net]

## Java Collections Example: max()

1. **import** java.util.\*;
2. **public** **class** CollectionsExample {
3. **public** **static** **void** main(String a[]){
4. List<Integer> list = **new** ArrayList<Integer>();
5. list.add(46);
6. list.add(67);
7. list.add(24);
8. list.add(16);
9. list.add(8);
10. list.add(12);
11. System.out.println("Value of maximum element from the collection: "+Collections.max(list));
12. }
13. }

Output:

Value of maximum element from the collection: 67

## Java Collections Example: min()

1. **import** java.util.\*;
2. **public** **class** CollectionsExample {
3. **public** **static** **void** main(String a[]){
4. List<Integer> list = **new** ArrayList<Integer>();
5. list.add(46);
6. list.add(67);
7. list.add(24);
8. list.add(16);
9. list.add(8);
10. list.add(12);
11. System.out.println("Value of minimum element from the collection: "+Collections.min(list));
12. }
13. }

# Sorting in Collection

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

|  |
| --- |
| **Collections** class provides static methods for sorting the elements of a collection. If collection elements are of a Set type, we can use TreeSet. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements. |

### Method of Collections class for sorting List elements

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

#### Note: String class and Wrapper classes implement the Comparable interface. So if you store the objects of string or wrapper classes, it will be Comparable.

### Example to sort string objects

1. **import** java.util.\*;
2. **class** TestSort1{
3. **public** **static** **void** main(String args[]){
5. ArrayList<String> al=**new** ArrayList<String>();
6. al.add("Viru");
7. al.add("Saurav");
8. al.add("Mukesh");
9. al.add("Tahir");
11. Collections.sort(al);
12. Iterator itr=al.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestSort1)

Mukesh

Saurav

Tahir

Viru

### Example to sort string objects in reverse order

1. **import** java.util.\*;
2. **class** TestSort2{
3. **public** **static** **void** main(String args[]){
5. ArrayList<String> al=**new** ArrayList<String>();
6. al.add("Viru");
7. al.add("Saurav");
8. al.add("Mukesh");
9. al.add("Tahir");
11. Collections.sort(al,Collections.reverseOrder());
12. Iterator i=al.iterator();
13. **while**(i.hasNext())
14. {
15. System.out.println(i.next());
16. }
17. }
18. }

Viru

Tahir

Saurav

Mukesh

### Example to sort Wrapper class objects

1. **import** java.util.\*;
2. **class** TestSort3{
3. **public** **static** **void** main(String args[]){
5. ArrayList al=**new** ArrayList();
6. al.add(Integer.valueOf(201));
7. al.add(Integer.valueOf(101));
8. al.add(230);//internally will be converted into objects as Integer.valueOf(230)
10. Collections.sort(al);
12. Iterator itr=al.iterator();
13. **while**(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. }
17. }

101

201

230

### Example to sort user-defined class objects

1. **import** java.util.\*;
3. **class** Student **implements** Comparable<Student> {
4. **public** String name;
5. **public** Student(String name) {
6. **this**.name = name;
7. }
8. **public** **int** compareTo(Student person) {
9. **return** name.compareTo(person.name);
11. }
12. }
13. **public** **class** TestSort4 {
14. **public** **static** **void** main(String[] args) {
15. ArrayList<Student> al=**new** ArrayList<Student>();
16. al.add(**new** Student("Viru"));
17. al.add(**new** Student("Saurav"));
18. al.add(**new** Student("Mukesh"));
19. al.add(**new** Student("Tahir"));
21. Collections.sort(al);
22. **for** (Student s : al) {
23. System.out.println(s.name);
24. }
25. }
26. }

Mukesh

Saurav

Tahir

Viru

# ava Comparable interface

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

### compareTo(Object obj) method

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

* positive integer, if the current object is greater than the specified object.
* negative integer, if the current object is less than the specified object.
* zero, if the current object is equal to the specified object.

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

### Collections class

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

### Method of Collections class for sorting List elements

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

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

## Java Comparable Example

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

*File: Student.java*

1. **class** Student **implements** Comparable<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. }
11. **public** **int** compareTo(Student st){
12. **if**(age==st.age)
13. **return** 0;
14. **else** **if**(age>st.age)
15. **return** 1;
16. **else**
17. **return** -1;
18. }
19. }

*File: TestSort1.java*

1. **import** java.util.\*;
2. **public** **class** TestSort1{
3. **public** **static** **void** main(String args[]){
4. ArrayList<Student> al=**new** ArrayList<Student>();
5. al.add(**new** Student(101,"Vijay",23));
6. al.add(**new** Student(106,"Ajay",27));
7. al.add(**new** Student(105,"Jai",21));
9. Collections.sort(al);
10. **for**(Student st:al){
11. System.out.println(st.rollno+" "+st.name+" "+st.age);
12. }
13. }
14. }

105 Jai 21

101 Vijay 23

106 Ajay 27

## Java Comparable Example: reverse order

Let's see the same example of the Comparable interface that sorts the list elements on the basis of age in reverse order.

*File: Student.java*

1. **class** Student **implements** Comparable<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. }
11. **public** **int** compareTo(Student st){
12. **if**(age==st.age)
13. **return** 0;
14. **else** **if**(age<st.age)
15. **return** 1;
16. **else**
17. **return** -1;
18. }
19. }

*File: TestSort2.java*

1. **import** java.util.\*;
2. **public** **class** TestSort2{
3. **public** **static** **void** main(String args[]){
4. ArrayList<Student> al=**new** ArrayList<Student>();
5. al.add(**new** Student(101,"Vijay",23));
6. al.add(**new** Student(106,"Ajay",27));
7. al.add(**new** Student(105,"Jai",21));
9. Collections.sort(al);
10. **for**(Student st:al){
11. System.out.println(st.rollno+" "+st.name+" "+st.age);
12. }
13. }
14. }

106 Ajay 27

101 Vijay 23

105 Jai 21

# Java Comparator interface

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

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

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

### Methods of Java Comparator Interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| public int compare(Object obj1, Object obj2) | It compares the first object with the second object. |
| public boolean equals(Object obj) | It is used to compare the current object with the specified object. |
| public boolean equals(Object obj) | It is used to compare the current object with the specified object. |

## Collections class

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

History of Java

#### Method of Collections class for sorting List elements

**public void sort(List list, Comparator c):** is used to sort the elements of List by the given Comparator.

## Java Comparator Example (Non-generic Old Style)

Let's see the example of sorting the elements of List on the basis of age and name. In this example, we have created 4 java classes:

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

## Java Comparator Example (Generic)

**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. }

**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.\*;
3. **class** Simple{
4. **public** **static** **void** main(String args[]){
6. ArrayList<Student> al=**new** ArrayList<Student>();
7. al.add(**new** Student(101,"Vijay",23));
8. al.add(**new** Student(106,"Ajay",27));
9. al.add(**new** Student(105,"Jai",21));
11. System.out.println("Sorting by Name");
13. Collections.sort(al,**new** NameComparator());
14. **for**(Student st: al){
15. System.out.println(st.rollno+" "+st.name+" "+st.age);
16. }
18. System.out.println("Sorting by age");
20. Collections.sort(al,**new** AgeComparator());
21. **for**(Student st: al){
22. System.out.println(st.rollno+" "+st.name+" "+st.age);
23. }
24. }
25. }

Sorting by Name

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by age

105 Jai 21

101 Vijay 23

106 Ajay 27