A HashMap in Java is a part of the Collections Framework that implements the Map interface.

It stores key-value pairs and allows for fast retrieval based on the key's hash code.

HashMap uses a hash table to store the pairs and provides constant-time performance for basic operations like get and put, assuming a good hash function that distributes keys uniformly across the buckets.

### . What is the Java Collections Framework?

The Java Collections Framework is a set of classes and interfaces that provide commonly used data structures and algorithms for manipulating collections of objects. It includes interfaces like List, Set, and Map, and classes like ArrayList, HashSet, and HashMap. The framework provides methods for inserting, deleting, and accessing elements, making it easier to manage groups of objects.

Collections in Java

1. [Java Collection Framework](https://www.javatpoint.com/collections-in-java)
2. [Hierarchy of Collection Framework](https://www.javatpoint.com/collections-in-java#collectionhierarchy)
3. [Collection interface](https://www.javatpoint.com/collections-in-java#collectionmethods)
4. [Iterator interface](https://www.javatpoint.com/collections-in-java#collectioniterator)

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet).

What is Collection in Java

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

What is a framework in Java

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

What is Collection framework

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

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

Do You Know?

* What are the two ways to iterate the elements of a collection?
* What is the difference between ArrayList and LinkedList classes in collection framework?
* What is the difference between ArrayList and Vector classes in collection framework?
* What is the difference between HashSet and HashMap classes in collection framework?
* What is the difference between HashMap and Hashtable class?
* What is the difference between Iterator and Enumeration interface in collection framework?
* How can we sort the elements of an object? What is the difference between Comparable and Comparator interfaces?
* What does the hashcode() method?
* What is the difference between Java collection and Java collections?

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. |

Iterator interface

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

Methods of Iterator interface

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

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

Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

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

1. Iterator<T> iterator()

It returns the iterator over the elements of type T.

Collection Interface

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

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

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

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

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

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

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

Output:

Ravi

Vijay

Ravi

Ajay

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

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

Output:

Ayush

Amit

Ashish

Garima

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

Consider the following example.

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

Output:

Ayush

Garvit

Amit

Ashish

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

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

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

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

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

Output:

head:Amit Sharma

head:Amit Sharma

iterating the queue elements:

Amit Sharma

Raj

JaiShankar

Vijay Raj

after removing two elements:

Raj

Vijay Raj

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

1. Deque d = **new** ArrayDeque();

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

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

Output:

Gautam

Karan

Ajay

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

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

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

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

Output:

Vijay

Ravi

Ajay

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

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

Output:

Ravi

Vijay

Ajay

SortedSet Interface

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

The SortedSet can be instantiated as:

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

TreeSet

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

Consider the following example:

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

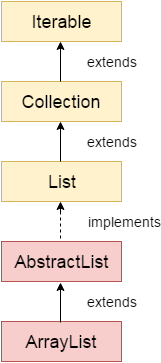
Output:

Ajay

Ravi

Vijay

Java ArrayList



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

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

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

The important points about the Java ArrayList class are:

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

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

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

Hierarchy of ArrayList class

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

ArrayList class declaration

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

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

Constructors of ArrayList

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

Methods of ArrayList

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

Java Non-generic Vs. Generic Collection

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

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

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

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

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

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

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

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

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. }

[**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.

**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. }

[**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

**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:**

[**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.

**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

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

**FileName:** ArrayList4.java

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.

**FileName:** ArrayList5.java

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

**Output:**

101 Sonoo 23

102 Ravi 21

103 Hanumat 25

Java ArrayList Serialization and Deserialization Example

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

**FileName:** ArrayList6.java

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

**Output:**

[Ravi, Vijay, Ajay]

Java ArrayList example to add elements

Here, we see different ways to add an element.

**FileName:** ArrayList7.java

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

**Output:**

Initial list of elements: []

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

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

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

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

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

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

Java ArrayList example to remove elements

Here, we see different ways to remove an element.

**FileName:** ArrayList8.java

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

**Output:**

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

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

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

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

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

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

After invoking clear() method: []

Java ArrayList example of retainAll() method

**FileName:** ArrayList9.java

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

**Output:**

iterating the elements after retaining the elements of al2

Ravi

Java ArrayList example of isEmpty() method

**FileName:** ArrayList4.java

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

**Output:**

Is ArrayList Empty: true

After Insertion

Is ArrayList Empty: false

Java ArrayList Example: Book

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

**FileName:** ArrayListExample20.java

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

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=ArrayListExample20)

**Output:**

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications and Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

Size and Capacity of an ArrayList

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

**FileName:** SizeCapacity.java

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

**Output:**

The size of the array is: 0

**Explanation:** The output makes sense as we have not done anything with the array list. Now observe the following program.

**FileName:** SizeCapacity1.java

1. **import** java.util.\*;
3. **public** **class** SizeCapacity1
4. {
6. **public** **static** **void** main(String[] args) **throws** Exception
7. {
9. ArrayList<Integer> al = **new** ArrayList<Integer>(10);
11. System.out.println("The size of the array is: " + al.size());
12. }
13. }

**Output:**

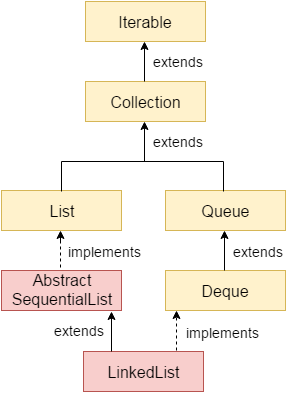
The size of the array is: 0

**Explanation:** We see that the size is still 0, and the reason behind this is the number 10 represents the capacity no the size. In fact, the size represents the total number of elements present in the array. As we have not added any element, therefore, the size of the array list is zero in both programs.

Capacity represents the total number of elements the array list can contain. Therefore, the capacity of an array list is always greater than or equal to the size of the array list. When we add an element to the array list, it checks whether the size of the array list has become equal to the capacity or not. If yes, then the capacity of the array list increases. So, in the above example, the capacity will be 10 till 10 elements are added to the list. When we add the 11th element, the capacity increases. Note that in both examples, the capacity of the array list is 10. In the first case, the capacity is 10 because the default capacity of the array list is 10. In the second case, we have explicitly mentioned that the capacity of the array list is 10.

Note: There is no any standard method to tell how the capacity increases in the array list. In fact, the way the capacity increases vary from one GDK version to the other version. Therefore, it is required to check the way capacity increases code is implemented in the GDK. There is no any pre-defined method in the ArrayList class that returns the capacity of the array list. Therefore, for better understanding, use the capacity() method of the Vector class. The logic of the size and the capacity is the same in the ArrayList class and the Vector class.

ava LinkedList class



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

The important points about Java LinkedList are:

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

Hierarchy of LinkedList class

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

Doubly Linked List

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



LinkedList class declaration

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

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

Constructors of Java LinkedList

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

Methods of Java LinkedList

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

Java LinkedList Example

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

Output: Ravi

Vijay

Ravi

Ajay

Java LinkedList example to add elements

Here, we see different ways to add elements.

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

Initial list of elements: []

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

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

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

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

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

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

After invoking addFirst(E e) method:

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

After invoking addLast(E e) method:

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

Java LinkedList example to remove elements

Here, we see different ways to remove an element.

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

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

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

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

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

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

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

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

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

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

After invoking clear() method: []

Java LinkedList Example to reverse a list of elements

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

Output: Ajay

Vijay

Ravi

Java LinkedList Example: Book

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

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

Difference Between ArrayList and LinkedList

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

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

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3) An ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |
| 5) The memory location for the elements of an ArrayList is contiguous. | The location for the elements of a linked list is not contagious. |
| 6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList. | | There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized. |
| 7) To be precise, an ArrayList is a resizable array. | LinkedList implements the doubly linked list of the list interface. |

Example of ArrayList and LinkedList in Java

Let's see a simple example where we are using ArrayList and LinkedList both.

**FileName:** TestArrayLinked.java

1. **import** java.util.\*;
2. **class** TestArrayLinked{
3. **public** **static** **void** main(String args[]){
5. List<String> al=**new** ArrayList<String>();//creating arraylist
6. al.add("Ravi");//adding object in arraylist
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
11. List<String> al2=**new** LinkedList<String>();//creating linkedlist
12. al2.add("James");//adding object in linkedlist
13. al2.add("Serena");
14. al2.add("Swati");
15. al2.add("Junaid");
17. System.out.println("arraylist: "+al);
18. System.out.println("linkedlist: "+al2);
19. }
20. }

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

**Output:**

arraylist: [Ravi,Vijay,Ravi,Ajay]

linkedlist: [James,Serena,Swati,Junaid]

Points to Remember

The following are some important points to remember regarding an ArrayList and LinkedList.

* When the rate of addition or removal rate is more than the read scenarios, then go for the LinkedList. On the other hand, when the frequency of the read scenarios is more than the addition or removal rate, then ArrayList takes precedence over LinkedList.
* Since the elements of an ArrayList are stored more compact as compared to a LinkedList; therefore, the ArrayList is more cache-friendly as compared to the LinkedList. Thus, chances for the cache miss are less in an ArrayList as compared to a LinkedList. Generally, it is considered that a LinkedList is poor in cache-locality.
* Memory overhead in the LinkedList is more as compared to the ArrayList. It is because, in a LinkedList, we have two extra links (next and previous) as it is required to store the address of the previous and the next nodes, and these links consume extra space. Such links are not present in an ArrayList.

Java List

**List** in Java provides the facility to maintain the *ordered collection*. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

List Interface declaration

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

Java List Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, E element) | It is used to insert the specified element at the specified position in a list. |
| boolean add(E e) | It is used to append the specified element at the end of a list. |
| boolean addAll(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the end of a list. |
| boolean addAll(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the specified position of the list. |
| void clear() | It is used to remove all of the elements from this list. |
|  |  |
| boolean equals(Object o) | It is used to compare the specified object with the elements of a list. |
| int hashcode() | It is used to return the hash code value for a list. |
| 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. |
| 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. |
| boolean contains(Object o) | It returns true if the list contains the specified element |
| boolean containsAll(Collection<?> c) | It returns true if the list contains all 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(Object o) | It is used to remove the first occurrence of the specified element. |  |
| boolean removeAll(Collection<?> c) | It is used to remove all the elements from the list. |  |
| void replaceAll(UnaryOperator<E> operator) | It is used to replace all the elements from the list with the specified element. |  |
| void retainAll(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. |  |

Java List vs ArrayList

List is an interface whereas ArrayList is the implementation class of List.

How to create List

The ArrayList and LinkedList classes provide the implementation of List interface. Let's see the examples to create the List:

1. //Creating a List of type String using ArrayList
2. List<String> list=**new** ArrayList<String>();
4. //Creating a List of type Integer using ArrayList
5. List<Integer> list=**new** ArrayList<Integer>();
7. //Creating a List of type Book using ArrayList
8. List<Book> list=**new** ArrayList<Book>();
10. //Creating a List of type String using LinkedList
11. List<String> list=**new** LinkedList<String>();

In short, you can create the List of any type. The ArrayList<T> and LinkedList<T> classes are used to specify the type. Here, T denotes the type.

Java List Example

Let's see a simple example of List where we are using the ArrayList class as the implementation.

1. **import** java.util.\*;
2. **public** **class** ListExample1{
3. **public** **static** **void** main(String args[]){
4. //Creating a List
5. List<String> list=**new** ArrayList<String>();
6. //Adding elements in the List
7. list.add("Mango");
8. list.add("Apple");
9. list.add("Banana");
10. list.add("Grapes");
11. //Iterating the List element using for-each loop
12. **for**(String fruit:list)
13. System.out.println(fruit);
15. }
16. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=ListExample1)

Output:

Mango

Apple

Banana

Grapes

How to convert Array to List

We can convert the Array to List by traversing the array and adding the element in list one by one using list.add() method. Let's see a simple example to convert array elements into List.

1. **import** java.util.\*;
2. **public** **class** ArrayToListExample{
3. **public** **static** **void** main(String args[]){
4. //Creating Array
5. String[] array={"Java","Python","PHP","C++"};
6. System.out.println("Printing Array: "+Arrays.toString(array));
7. //Converting Array to List
8. List<String> list=**new** ArrayList<String>();
9. **for**(String lang:array){
10. list.add(lang);
11. }
12. System.out.println("Printing List: "+list);
14. }
15. }

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

Output:

Printing Array: [Java, Python, PHP, C++]

Printing List: [Java, Python, PHP, C++]

How to convert List to Array

We can convert the List to Array by calling the list.toArray() method. Let's see a simple example to convert list elements into array.

1. **import** java.util.\*;
2. **public** **class** ListToArrayExample{
3. **public** **static** **void** main(String args[]){
4. List<String> fruitList = **new** ArrayList<>();
5. fruitList.add("Mango");
6. fruitList.add("Banana");
7. fruitList.add("Apple");
8. fruitList.add("Strawberry");
9. //Converting ArrayList to Array
10. String[] array = fruitList.toArray(**new** String[fruitList.size()]);
11. System.out.println("Printing Array: "+Arrays.toString(array));
12. System.out.println("Printing List: "+fruitList);
13. }
14. }

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

Output:

Printing Array: [Mango, Banana, Apple, Strawberry]

Printing List: [Mango, Banana, Apple, Strawberry]

Get and Set Element in List

The *get() method* returns the element at the given index, whereas the *set() method* changes or replaces the element.

1. **import** java.util.\*;
2. **public** **class** ListExample2{
3. **public** **static** **void** main(String args[]){
4. //Creating a List
5. List<String> list=**new** ArrayList<String>();
6. //Adding elements in the List
7. list.add("Mango");
8. list.add("Apple");
9. list.add("Banana");
10. list.add("Grapes");
11. //accessing the element
12. System.out.println("Returning element: "+list.get(1));//it will return the 2nd element, because index starts from 0
13. //changing the element
14. list.set(1,"Dates");
15. //Iterating the List element using for-each loop
16. **for**(String fruit:list)
17. System.out.println(fruit);
19. }
20. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=ListExample2)

**Output:**

Returning element: Apple

Mango

Dates

Banana

Grapes

How to Sort List

There are various ways to sort the List, here we are going to use Collections.sort() method to sort the list element. The *java.util* package provides a utility class **Collections** which has the static method sort(). Using the **Collections.sort()** method, we can easily sort any List.

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 ListIterator Interface

ListIterator Interface is used to traverse the element in a backward and forward direction.

ListIterator Interface declaration

1. **public** **interface** ListIterator<E> **extends** Iterator<E>

Methods of Java ListIterator Interface:

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(E e) | This method inserts the specified element into the list. |
| boolean hasNext() | This method returns true if the list iterator has more elements while traversing the list in the forward direction. |
| E next() | This method returns the next element in the list and advances the cursor position. |
| int nextIndex() | This method returns the index of the element that would be returned by a subsequent call to next() |
| boolean hasPrevious() | This method returns true if this list iterator has more elements while traversing the list in the reverse direction. |
| E previous() | This method returns the previous element in the list and moves the cursor position backward. |
| E previousIndex() | This method returns the index of the element that would be returned by a subsequent call to previous(). |
| void remove() | This method removes the last element from the list that was returned by next() or previous() methods |
| void set(E e) | This method replaces the last element returned by next() or previous() methods with the specified element. |

Example of ListIterator Interface

1. **import** java.util.\*;
2. **public** **class** ListIteratorExample1{
3. **public** **static** **void** main(String args[]){
4. List<String> al=**new** ArrayList<String>();
5. al.add("Amit");
6. al.add("Vijay");
7. al.add("Kumar");
8. al.add(1,"Sachin");
9. ListIterator<String> itr=al.listIterator();
10. System.out.println("Traversing elements in forward direction");
11. **while**(itr.hasNext()){
13. System.out.println("index:"+itr.nextIndex()+" value:"+itr.next());
14. }
15. System.out.println("Traversing elements in backward direction");
16. **while**(itr.hasPrevious()){
18. System.out.println("index:"+itr.previousIndex()+" value:"+itr.previous());
19. }
20. }
21. }

Output:

Traversing elements in forward direction

index:0 value:Amit

index:1 value:Sachin

index:2 value:Vijay

index:3 value:Kumar

Traversing elements in backward direction

index:3 value:Kumar

index:2 value:Vijay

index:1 value:Sachin

index:0 value:Amit

Example of List: Book

Let's see an example of List where we are adding the Books.

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ListExample5 {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** ArrayList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications and Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }
31. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=ListExample5)

Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications and Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

Java HashSet



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

The important points about Java HashSet class are:

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

Difference between List and Set

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

Hierarchy of HashSet class

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

HashSet class declaration

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

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

Constructors of Java HashSet class

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

Methods of Java HashSet class

Various methods of Java HashSet class are as follows:

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

Java HashSet Example

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

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

Five

One

Four

Two

Three

Java HashSet example ignoring duplicate elements

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

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

Ajay

Vijay

Ravi

Java HashSet example to remove elements

Here, we see different ways to remove an element.

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

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

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

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

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

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

After invoking clear() method: []

Java HashSet from another Collection

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

Vijay

Ravi

Gaurav

Ajay

Java HashSet Example: Book

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

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

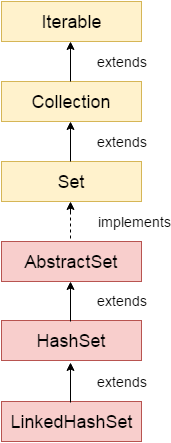
Output:

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

Java LinkedHashSet Class



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

The important points about the Java LinkedHashSet class are:

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

Note: Keeping the insertion order in the LinkedHashset has some additional costs, both in terms of extra memory and extra CPU cycles. Therefore, if it is not required to maintain the insertion order, go for the lighter-weight HashMap or the HashSet instead.

Hierarchy of LinkedHashSet class

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

LinkedHashSet Class Declaration

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

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

Constructors of Java LinkedHashSet Class

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

Java LinkedHashSet Example

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

**FileName:** LinkedHashSet1.java

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

**Output:**

One

Two

Three

Four

Five

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

Java LinkedHashSet example ignoring duplicate Elements

**sFileName:** LinkedHashSet2.java

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

**Output:**

Ravi

Vijay

Ajay

Remove Elements Using LinkeHashSet Class

**FileName:** LinkedHashSet3.java

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

**Output:**

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

true

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

false

Java LinkedHashSet Example: Book

**FileName:** Book.java

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

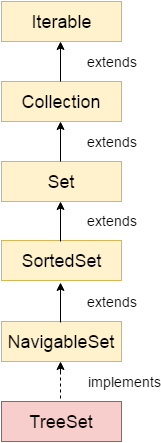
**Output:**

101 Let us C Yashwant Kanetkar BPB 8

102 Data Communications & Networking Forouzan Mc Graw Hill 4

103 Operating System Galvin Wiley 6

Java TreeSet class



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

The important points about the Java TreeSet class are:

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

Internal Working of The TreeSet Class

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

Synchronization of The TreeSet Class

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

1. TreeSet treeSet = **new** TreeSet();
2. Set syncrSet = Collections.synchronziedSet(treeSet);

Hierarchy of TreeSet class

As shown in the above diagram, the 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.

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 a comparator that arranges elements in order. |
| Iterator descendingIterator() | It is used to iterate the elements in descending order. |
| NavigableSet descendingSet() | It returns the elements in reverse order. |
| E floor(E e) | It returns the equal or closest least element of the specified element from the set, or null there is no such element. |
| SortedSet headSet(E toElement) | It returns the group of elements that are less than the specified element. |
| NavigableSet headSet(E toElement, boolean inclusive) | It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element. |
| E higher(E e) | It returns the closest greatest element of the specified element from the set, or null there is no such element. |
| Iterator iterator() | It is used to iterate the elements in ascending order. |
| E lower(E e) | It returns the closest least element of the specified element from the set, or null there is no such element. |
| E pollFirst() | It is used to retrieve and remove the lowest(first) element. |
| E pollLast() | It is used to retrieve and remove the highest(last) element. |
| Spliterator spliterator() | It is used to create a late-binding and fail-fast spliterator over the elements. |
| NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) | It returns a set of elements that lie between the given range. |
| SortedSet subSet(E fromElement, E toElement)) | It returns a set of elements that lie between the given range which includes fromElement and excludes toElement. |
| SortedSet tailSet(E fromElement) | It returns a set of elements that are greater than or equal to the specified element. |
| NavigableSet tailSet(E fromElement, boolean inclusive) | It returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element. |
| boolean contains(Object o) | It returns true if this set contains the specified element. |
| boolean isEmpty() | It returns true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| void clear() | It is used to remove all of the elements from this set. |
| Object clone() | It returns a shallow copy of this TreeSet instance. |
| E first() | It returns the first (lowest) element currently in this sorted set. |
| E last() | It returns the last (highest) element currently in this sorted set. |
| int size() | It returns the number of elements in this set. |

Java TreeSet Examples

Java TreeSet Example 1:

Let's see a simple example of Java TreeSet.

**FileName:** TreeSet1.java

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.

**FileName:** TreeSet2.java

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

Java TreeSet Example 3:

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

**FileName:** TreeSet3.java

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

**Output:**

Lowest Value: 12

Highest Value: 66

Java TreeSet Example 4:

In this example, we perform various NavigableSet operations.

**FileName:** TreeSet4.java

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

**Output:**

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

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

Head Set: [A, B, C]

SubSet: [B, C, D, E]

TailSet: [D, E]

Java TreeSet Example 5:

In this example, we perform various SortedSetSet operations.

**FileName:** TreeSet5.java

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

**Output:**

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

Head Set: [A, B]

SubSet: [A, B, C, D]

TailSet: [C, D, E]

Java TreeSet Example: Book

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

**FileName:** TreeSetExample.java

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

**Output:**

101 Data Communications & Networking Forouzan Mc Graw Hill 4

121 Let us C Yashwant Kanetkar BPB 8

233 Operating System Galvin Wiley 6

ClassCast Exception in TreeSet

If we add an object of the class that is not implementing the Comparable interface, the ClassCast Exception is raised. Observe the following program.

**FileName:** ClassCastExceptionTreeSet.java

1. // important import statement
2. **import** java.util.\*;
4. **class** Employee
5. {
7. **int** empId;
8. String name;
10. // getting the name of the employee
11. String getName()
12. {
13. **return** **this**.name;
14. }
16. // setting the name of the employee
17. **void** setName(String name)
18. {
19. **this**.name = name;
20. }
22. // setting the employee id
23. // of the employee
24. **void** setId(**int** a)
25. {
26. **this**.empId = a;
27. }
29. // retrieving the employee id of
30. // the employee
31. **int** getId()
32. {
33. **return** **this**.empId;
34. }
36. }
38. **public** **class** ClassCastExceptionTreeSet
39. {
41. // main method
42. **public** **static** **void** main(String[] argvs)
43. {
44. // creating objects of the class Employee
45. Employee obj1 = **new** Employee();
47. Employee obj2 = **new** Employee();
49. TreeSet<Employee> ts =  **new** TreeSet<Employee>();
51. // adding the employee objects to
52. // the TreeSet class
53. ts.add(obj1);
54. ts.add(obj2);
56. System.out.println("The program has been executed successfully.");
58. }
59. }

When we compile the above program, we get the ClassCastException, as shown below.

Exception in thread "main" java.lang.ClassCastException: class Employee cannot be cast to class java.lang.Comparable (Employee is in unnamed module of loader 'app'; java.lang.Comparable is in module java.base of loader 'bootstrap')

at java.base/java.util.TreeMap.compare(TreeMap.java:1569)

at java.base/java.util.TreeMap.addEntryToEmptyMap(TreeMap.java:776)

at java.base/java.util.TreeMap.put(TreeMap.java:785)

at java.base/java.util.TreeMap.put(TreeMap.java:534)

at java.base/java.util.TreeSet.add(TreeSet.java:255)

at ClassCastExceptionTreeSet.main(ClassCastExceptionTreeSet.java:52)

**Explanation:** In the above program, it is required to implement a Comparable interface. It is because the TreeSet maintains the sorting order, and for doing the sorting the comparison of different objects that are being inserted in the TreeSet is must, which is accomplished by implementing the Comparable interface.

Java Map Interface

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

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

Java Map Hierarchy

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

Java Map Hierarchy

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

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

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

Useful methods of Map interface

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

Map.Entry Interface

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

Methods of Map.Entry interface

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

Java Map Example: Non-Generic (Old Style)

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

Output:

1 Amit

2 Jai

5 Rahul

6 Amit

Java Map Example: Generic (New Style)

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

Output:

102 Rahul

100 Amit

101 Vijay

Java Map Example: comparingByKey()

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

Output:

100=Amit

101=Vijay

102=Rahul

Java Map Example: comparingByKey() in Descending Order

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

Output:

102=Rahul

101=Vijay

100=Amit

Java Map Example: comparingByValue()

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

Output:

100=Amit

102=Rahul

101=Vijay

Java Map Example: comparingByValue() in Descending Order

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

Output:

101=Vijay

102=Rahul

100=Amit

Java HashMap



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

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

Points to remember

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

Hierarchy of HashMap class

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

HashMap class declaration

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

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

HashMap class Parameters

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

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

Constructors of Java HashMap class

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

Methods of Java HashMap class

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

Java HashMap Example

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

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

[**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)

Iterating Hashmap...

1 Grapes

2 Apple

3 Banana

Java HashMap example to add() elements

Here, we see different ways to insert elements.

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

Initial list of elements: {}

After invoking put() method

100 Amit

101 Vijay

102 Rahul

After invoking putIfAbsent() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

After invoking putAll() method

100 Amit

101 Vijay

102 Rahul

103 Gaurav

104 Ravi

Java HashMap example to remove() elements

Here, we see different ways to remove elements.

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

Output:

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

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

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

Updated list of elements: {103=Gaurav}

Java HashMap example to replace() elements

Here, we see different ways to replace elements.

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

Initial list of elements:

100 Amit

101 Vijay

102 Rahul

Updated list of elements:

100 Amit

101 Vijay

102 Gaurav

Updated list of elements:

100 Amit

101 Ravi

102 Gaurav

Updated list of elements:

100 Ajay

101 Ajay

102 Ajay

Difference between HashSet and HashMap

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

Java 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)

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

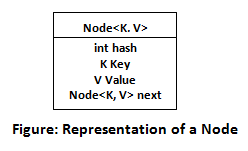
Working of HashMap in Java

What is Hashing

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

What is HashMap

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.



Before understanding the internal working of HashMap, you must be aware of hashCode() and equals() method.

* **equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.
* **hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.
* **Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.



Insert Key, Value pair in HashMap

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

Example

In the following example, we want to insert three (Key, Value) pair in the HashMap.

1. HashMap<String, Integer> map = **new** HashMap<>();
2. map.put("Aman", 19);
3. map.put("Sunny", 29);
4. map.put("Ritesh", 39);

Let's see at which index the Key, value pair will be saved into HashMap. When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

Calculating Index

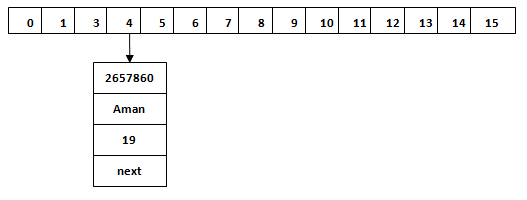
Index minimizes the size of the array. The Formula for calculating the index is:

1. Index = hashcode(Key) & (n-1)

Where n is the size of the array. Hence the index value for "Aman" is:

1. Index = 2657860 & (16-1) = 4

The value 4 is the computed index value where the Key and value will store in HashMap.

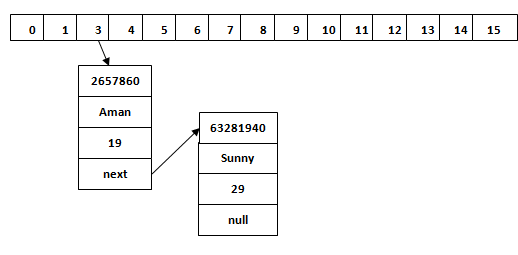


Hash Collision

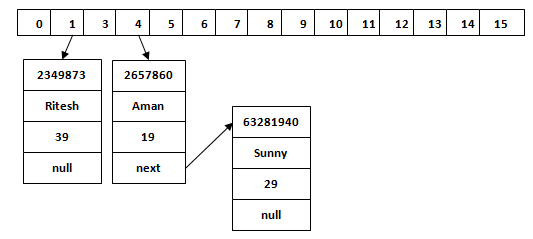
This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

1. Index=63281940 & (16-1) = 4

The value 4 is the computed index value where the Key will be stored in HashMap. In this case, equals() method check that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 4.



Similarly, we will store the Key "Ritesh." Suppose hash code for the Key is 2349873. The index value will be 1. Hence this Key will be stored at index 1.



get() method in HashMap

get() method is used to get the value by its Key. It will not fetch the value if you don't know the Key. When get(K Key) method is called, it calculates the hash code of the Key.

Suppose we have to fetch the Key "Aman." The following method will be called.

1. map.get(**new** Key("Aman"));

It generates the hash code as 2657860. Now calculate the index value of 2657860 by using index formula. The index value will be 4, as we have calculated above. get() method search for the index value 4. It compares the first element Key with the given Key. If both keys are equal, then it returns the value else check for the next element in the node if it exists. In our scenario, it is found as the first element of the node and return the value 19.

Let's fetch another Key "Sunny."

The hash code of the Key "Sunny" is 63281940. The calculated index value of 63281940 is 4, as we have calculated for put() method. Go to index 4 of the array and compare the first element's Key with the given Key. It also compares Keys. In our scenario, the given Key is the second element, and the next of the node is null. It compares the second element Key with the specified Key and returns the value 29. It returns null if the next of the node is null.

Java LinkedHashMap class



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

Points to remember

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

LinkedHashMap class declaration

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

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

LinkedHashMap class Parameters

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

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

Constructors of Java LinkedHashMap class

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

Methods of Java LinkedHashMap class

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

Java LinkedHashMap Example

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

Output:100 Amit

101 Vijay

102 Rahul

Java LinkedHashMap Example: Key-Value pair

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

Keys: [100, 101, 102]

Values: [Amit, Vijay, Rahul]

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

Java LinkedHashMap Example:remove()

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

Output:

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

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

Java LinkedHashMap 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** LinkedHashMap<Integer,Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to map
23. map.put(2,b2);
24. map.put(1,b1);
25. map.put(3,b3);
27. //Traversing map
28. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
29. **int** key=entry.getKey();
30. Book b=entry.getValue();
31. System.out.println(key+" Details:");
32. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
33. }
34. }
35. }

Output:

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

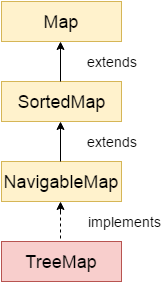
1 Details:

101 Let us C Yashwant Kanetkar BPB 8

3 Details:

103 Operating System Galvin Wiley 6

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.

TreeMap class Parameters

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

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

Constructors of Java TreeMap class

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

Methods of Java TreeMap class

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

Java TreeMap Example

Output:100 Amit

101 Vijay

102 Ravi

103 Rahul

Java TreeMap Example: remove()

Output:

Before invoking remove() method

100 Amit

101 Vijay

102 Ravi

103 Rahul

After invoking remove() method

100 Amit

101 Vijay

103 Rahul

Java TreeMap Example: NavigableMap

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

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

tailMap: {102=Ravi, 103=Rahul}

subMap: {101=Vijay, 102=Ravi}

Java TreeMap Example: SortedMap

headMap: {100=Amit, 101=Vijay}

tailMap: {102=Ravi, 103=Rahul}

subMap: {100=Amit, 101=Vijay}

What is difference between HashMap and TreeMap?

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

Java TreeMap Example: Book

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

Java Hashtable class

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

Points to remember

* A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* Java Hashtable class contains unique elements.
* Java Hashtable class doesn't allow null key or value.
* Java Hashtable class is synchronized.
* The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Hashtable class declaration

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

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

Hashtable class Parameters

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

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

Constructors of Java Hashtable class

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

Methods of Java Hashtable class

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

Java Hashtable Example

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

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestCollection16)

Output:

103 Rahul

102 Ravi

101 Vijay

100 Amit

Java Hashtable Example: remove()

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

Output:

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

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

Java Hashtable Example: getOrDefault()

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

Output:

Vijay

Not Found

Java Hashtable Example: putIfAbsent()

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

Output:

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

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

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

Java Hashtable 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** HashtableExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating map of Books
17. Map<Integer,Book> map=**new** Hashtable<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);
26. //Traversing map
27. **for**(Map.Entry<Integer, Book> entry:map.entrySet()){
28. **int** key=entry.getKey();
29. Book b=entry.getValue();
30. System.out.println(key+" Details:");
31. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
32. }
33. }
34. }

Output:

3 Details:

103 Operating System Galvin Wiley 6

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

Difference between HashMap and Hashtable

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

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

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

Java Comparable interface

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

compareTo(Object obj) method

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

* 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.

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

Java 8 Comparator interface

Java 8 Comparator interface is a functional interface that contains only one abstract method. Now, we can use the Comparator interface as the assignment target for a lambda expression or method reference.

Methods of Java 8 Comparator Interface

|  |
| --- |
|  |

|  |  |
| --- | --- |
| **Method** | **Description** |
| int compare(T o1, T o2) | It compares the first object with second object. |
| static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor) | It accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator that compares by that sort key. |
| static <T,U> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator) | It accepts a function that extracts a sort key from a type T, and returns a Comparator that compares by that sort key using the specified Comparator. |
| static <T> Comparator<T> comparingDouble(ToDoubleFunction<? super T> keyExtractor) | It accepts a function that extracts a double sort key from a type T, and returns a Comparator that compares by that sort key. |
| static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor) | It accepts a function that extracts an int sort key from a type T, and returns a Comparator that compares by that sort key. |
| static <T> Comparator<T> comparingLong(ToLongFunction<? super T> keyExtractor) | It accepts a function that extracts a long sort key from a type T, and returns a Comparator that compares by that sort key. |
| boolean equals(Object obj) | It is used to compare the current object with the specified object. |
| static <T extends Comparable<? super T>> Comparator<T> naturalOrder() | It returns a comparator that compares Comparable objects in natural order. |
| static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator) | It returns a comparator that treats null to be less than non-null elements. |
| static <T> Comparator<T> nullsLast(Comparator<? super T> comparator) | It returns a comparator that treats null to be greater than non-null elements. |
| default Comparator<T> reversed() | It returns comparator that contains reverse ordering of the provided comparator. |
| static <T extends Comparable<? super T>> Comparator<T> reverseOrder() | It returns comparator that contains reverse of natural ordering. |
| default Comparator<T> thenComparing(Comparator<? super T> other) | It returns a lexicographic-order comparator with another comparator. |
| default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor) | It returns a lexicographic-order comparator with a function that extracts a Comparable sort key. |
| default <U> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator) | It returns a lexicographic-order comparator with a function that extracts a key to be compared with the given Comparator. |
| default Comparator<T> thenComparingDouble(ToDoubleFunction<? super T> keyExtractor) | It returns a lexicographic-order comparator with a function that extracts a double sort key. |
| default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor) | It returns a lexicographic-order comparator with a function that extracts a int sort key. |
| default Comparator<T> thenComparingLong(ToLongFunction<? super T> keyExtractor) | It returns a lexicographic-order comparator with a function that extracts a long sort key. |

Java 8 Comparator Example

Let's see the example of sorting the elements of List on the basis of age and name.

*File: 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. }
11. **public** **int** getRollno() {
12. **return** rollno;
13. }
15. **public** **void** setRollno(**int** rollno) {
16. **this**.rollno = rollno;
17. }
19. **public** String getName() {
20. **return** name;
21. }
23. **public** **void** setName(String name) {
24. **this**.name = name;
25. }
27. **public** **int** getAge() {
28. **return** age;
29. }
31. **public** **void** setAge(**int** age) {
32. **this**.age = age;
33. }
35. }

*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));
8. /Sorting elements on the basis of name
9. Comparator<Student> cm1=Comparator.comparing(Student::getName);
10. Collections.sort(al,cm1);
11. System.out.println("Sorting by Name");
12. **for**(Student st: al){
13. System.out.println(st.rollno+" "+st.name+" "+st.age);
14. }
15. //Sorting elements on the basis of age
16. Comparator<Student> cm2=Comparator.comparing(Student::getAge);
17. Collections.sort(al,cm2);
18. System.out.println("Sorting by Age");
19. **for**(Student st: al){
20. System.out.println(st.rollno+" "+st.name+" "+st.age);
21. }
22. }
23. }

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 8 Comparator Example: nullsFirst() and nullsLast() method

Here, we sort the list of elements that also contains null.

*File: 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. **public** **int** getRollno() {
11. **return** rollno;
12. }
13. **public** **void** setRollno(**int** rollno) {
14. **this**.rollno = rollno;
15. }
16. **public** String getName() {
17. **return** name;
18. }
20. **public** **void** setName(String name) {
21. **this**.name = name;
22. }
24. **public** **int** getAge() {
25. **return** age;
26. }
27. **public** **void** setAge(**int** age) {
28. **this**.age = age;
29. }
30. }

*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,**null**,21));
8. Comparator<Student> cm1=Comparator.comparing(Student::getName,Comparator.nullsFirst(String::compareTo));
9. Collections.sort(al,cm1);
10. System.out.println("Considers null to be less than non-null");
11. **for**(Student st: al){
12. System.out.println(st.rollno+" "+st.name+" "+st.age);
13. }
14. Comparator<Student> cm2=Comparator.comparing(Student::getName,Comparator.nullsLast(String::compareTo));
15. Collections.sort(al,cm2);
16. System.out.println("Considers null to be greater than non-null");
17. **for**(Student st: al){
18. System.out.println(st.rollno+" "+st.name+" "+st.age);
19. }
20. }
21. }

Considers null to be less than non-null

105 null 21

106 Ajay 27

101 Vijay 23

Considers null to be greater than non-null

106 Ajay 27

101 Vijay 23

105 null 21

Difference between Comparable and Comparator

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

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

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

Java Comparable Example

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

*File: TestSort3.java*

1. //Java Program to demonstrate the use of Java Comparable.
2. //Creating a class which implements Comparable Interface
3. **import** java.util.\*;
4. **import** java.io.\*;
5. **class** Student **implements** Comparable<Student>{
6. **int** rollno;
7. String name;
8. **int** age;
9. Student(**int** rollno,String name,**int** age){
10. **this**.rollno=rollno;
11. **this**.name=name;
12. **this**.age=age;
13. }
14. **public** **int** compareTo(Student st){
15. **if**(age==st.age)
16. **return** 0;
17. **else** **if**(age>st.age)
18. **return** 1;
19. **else**
20. **return** -1;
21. }
22. }
23. //Creating a test class to sort the elements
24. **public** **class** TestSort3{
25. **public** **static** **void** main(String args[]){
26. ArrayList<Student> al=**new** ArrayList<Student>();
27. al.add(**new** Student(101,"Vijay",23));
28. al.add(**new** Student(106,"Ajay",27));
29. al.add(**new** Student(105,"Jai",21));
31. Collections.sort(al);
32. **for**(Student st:al){
33. System.out.println(st.rollno+" "+st.name+" "+st.age);
34. }
35. }
36. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestSort3)

Output:

105 Jai 21

101 Vijay 23

106 Ajay 27

Java Comparator Example

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

**Student.java**

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

**AgeComparator.java**

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

**NameComparator.java**

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

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

**TestComparator.java**

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

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

Output:

Sorting by Name

106 Ajay 27

105 Jai 21

101 Vijay 23

Sorting by Age

105 Jai 21

101 Vijay 23

106 Ajay 27

Java ConcurrentHashMap class

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

Java ConcurrentHashMap class declaration

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

List of ConcurrentHashMap class Methods

|  |  |  |
| --- | --- | --- |
| **NO** | **Method** | **Description** |
| 1. | public void [clear()](https://www.javatpoint.com/java-concurrenthashmap-clear-method) | The clear() method of ConcurrentHashMap class removes all of the mappings from this map. |
| 2. | public V [compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)](https://www.javatpoint.com/java-concurrenthashmap-compute-method) | The compute() method of ConcurrentHashMap class Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| 3. | public V [computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)](https://www.javatpoint.com/java-concurrenthashmap-computeifabsent-method) | The computeIfAbsent() method of ConcurrentHashMap class attempts to compute its value using the given mapping function and enters it into this map unless null If the specified key is not already associated with a value. |
| 4. | public V [computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)](https://www.javatpoint.com/java-concurrenthashmap-computeifpresent-method) | The computeIfPresent() method of ConcurrentHashMap class Attempts to compute a new mapping given the key and its current mapped value, If the value for the specified key is present. |
| 5. | public boolean [contains(Object value)](https://www.javatpoint.com/java-concurrenthashmap-contains-method) | The contains() method of ConcurrentHashMap class tests if some key maps into the specified value in this table.. |
| 6. | public boolean [containsKey(Object key)](https://www.javatpoint.com/java-concurrenthashmap-containskey-method) | The containsKey() method of ConcurrentHashMap class tests if the specified object is a key in this table. |
| 7. | public boolean [containsValue(Object value)](https://www.javatpoint.com/java-concurrenthashmap-containsvalue-method) | The containsValue() method of ConcurrentHashMap class returns true if this map maps one or more keys to the specified value. Note: This method may require a full traversal of the map, and is much slower than method containsKey. |
| 8. | public Enumeration<V> [elements()](https://www.javatpoint.com/java-concurrenthashmap-elements-method) | The elements() method of ConcurrentHashMap class returns an enumeration of the values in this table. |
| 9. | public Set<Map.Entry<K,V>> [entrySet()](https://www.javatpoint.com/java-concurrenthashmap-entryset-method) | The entrySet() method of ConcurrentHashMap class Returns a Set view of the mappings contained in this map. The changes made to the map are reflected in the set, and vice-versa. |
| 10. | public boolean [equals(Object o)](https://www.javatpoint.com/java-concurrenthashmap-equals-method) | The elements() method of ConcurrentHashMap class Compares the specified object with this map for equality and returns true if the given object is a map with the same mappings as this map. |
| 11. | public V [get(Object key)](https://www.javatpoint.com/java-concurrenthashmap-get-method) | The get() method of ConcurrentHashMap class Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. |
| 12. | public V [getOrDefault(Object key, V defaultValue)](https://www.javatpoint.com/java-concurrenthashmap-getordefault-method) | The getOrDefault() method of ConcurrentHashMap class Returns the value to which the specified key is mapped, or the given default value if this map contains no mapping for the key. |
| 13. | public int [hashCode()](https://www.javatpoint.com/java-concurrenthashmap-hashcode-method) | The hashcode() method of ConcurrentHashMap class Returns the hash code value for this Map, i.e., the sum of, for each key-value pair in the map, key.hashCode() ^ value.hashCode(). |
| 14. | public Enumeration<K> [keys()](https://www.javatpoint.com/java-concurrenthashmap-keys-method) | The keys() method of ConcurrentHashMap class returns an enumeration of the keys in this table. |
| 15. | public ConcurrentHashMap.KeySetView<K,V> [keySet()](https://www.javatpoint.com/java-concurrenthashmap-keyset-method) public ConcurrentHashMap.KeySetView<K,V> [keySet(V mappedValue)](https://www.javatpoint.com/java-concurrenthashmap-keyset-method) | The keySet() method of ConcurrentHashMap class returns a Set view of the keys contained in this map. The set is stacked by the map, so changes to the map are reflected in the set, and vice-versa. |
| 16. | public long [mappingCount()](https://www.javatpoint.com/java-concurrenthashmap-mappingcount-method) | The mappingCount() method of ConcurrentHashMap class returns the number of mappings. The value returned is an estimated value; the actual count may differ if there are concurrent insertions or removals. |
| 17. | public V [merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)](https://www.javatpoint.com/java-concurrenthashmap-merge-method) | The merge() method of ConcurrentHashMap class merge sets If the specified key is not already associated with a (non-null) value, associates it with the given value. |
| 18. | public static <K> ConcurrentHashMap.KeySetView<K,Boolean> [newKeySet()](https://www.javatpoint.com/java-concurrenthashmap-newkeyset-method) public static <K> ConcurrentHashMap.KeySetView<K,Boolean> [newKeySet(int initialCapacity)](https://www.javatpoint.com/java-concurrenthashmap-newkeyset-method) | The newKeySet() method of ConcurrentHashMap class Creates a new Set backed by a ConcurrentHashMap from the given type to Boolean.TRUE. |
| 19. | public V [put(K key, V value)](https://www.javatpoint.com/java-concurrenthashmap-put-method) | The put() method of ConcurrentHashMap class Maps the specified key to the specified value in this table. |
| 20. | public void [putAll(Map<? extends K,? extends V> m)](https://www.javatpoint.com/java-concurrenthashmap-putall-method) | The putAll() method of ConcurrentHashMap class Copies all of the mappings from the specified map to this one. These mappings replace any mappings that this map had for any of the keys currently in the specified map. |
| 21. | public V [putIfAbsent(K key, V value)](https://www.javatpoint.com/java-concurrenthashmap-putifabsent-method) | The putIfAbsent() method of ConcurrentHashMap class Maps If the specified key is not already associated with a value, associates it with the given value. This is equivalent to, for this map. |
| 22. | public V [remove(Object key)](https://www.javatpoint.com/java-concurrenthashmap-remove-method) public boolean [remove(Object key, Object value)](https://www.javatpoint.com/java-concurrenthashmap-remove-method) | The remove() method of ConcurrentHashMap class Removes the key (and its corresponding value) from this map. This method does nothing if the key is not on the map. |
| 23. | public V [replace(K key, V value)](https://www.javatpoint.com/java-concurrenthashmap-replace-method) public boolean [replace(K key, V oldValue, V newValue)](https://www.javatpoint.com/java-concurrenthashmap-replace-method) | The replace() method of ConcurrentHashMap class replaces the entry for a key only if currently mapped to some value. This is equivalent to, for this map. |
| 24. | public String [toString()](https://www.javatpoint.com/java-concurrenthashmap-tostring-method) | The toString() method of ConcurrentHashMap class returns a string representation of this map. The string representation consists of a list of key-value mappings (in no particular order) enclosed in braces ("{}"). |
| 25. | public void [forEach(long parallelismThreshold, BiConsumer<? super K,? super V> action)](https://www.javatpoint.com/java-concurrenthashmap-foreach-method) public <U> void [forEach(long parallelismThreshold, BiFunction<? super K,? super V,? extends U> transformer, Consumer<? super U> action)](https://www.javatpoint.com/java-concurrenthashmap-foreach-method) | The forEach() method of ConcurrentHashMap class Performs the given action for each (key, value). |
| 26. | public Collection<V> [values()](https://www.javatpoint.com/java-concurrenthashmap-values-method) | The values() method of ConcurrentHashMap class returns a Collection view of the values contained in this map. The map backs the collection, so changes to the map are reflected in the collection, and vice-versa. The collection supports element removal, which removes the corresponding mapping from this map, via the Iterator |

Java ConcurrentHashMap class Example: computeIfAbsent()

1. //import statement
3. **import** java.util.concurrent.\*;
5. **import** java.util.\*;
7. **public** **class** ConcurrentHashMapcomputeIfAbsentExample1\_1 {
9. **public** **static** **void** main(String[] args)
10. {
11. // crete a HashMap and add some values
12. HashMap<String, Integer> mapcon
13. = **new** HashMap<>();
14. mapcon.put("k1", 100);
15. mapcon.put("k2", 200);
16. mapcon.put("k3", 300);
17. mapcon.put("k4", 400);
18. System.out.println("HashMap values :\n " + mapcon.toString());
19. mapcon.computeIfAbsent("k5", k -> 200 + 300);
20. mapcon.computeIfAbsent("k6", k -> 60 \* 10);
21. System.out.println("New HashMap after computeIfAbsent :\n "+ mapcon);
22. }
23. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=ConcurrentHashMapcomputeIfAbsentExample1_1)

**Output:**

HashMap values :

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

New HashMap after computeIfAbsent :

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

Java ConcurrentHashMap Class Example: containsValue()

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

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=ConcurrentHashMapclearExample1)

**Output:**

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

is 255 present? :: true