# Collections in Java

* **The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.**
* Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
* **Java Collection means a single unit of objects.**
* Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet).

|  |
| --- |
| Adding 150 Student names :========================  String name1,name2,name3……………………….etc.  As this is a tedious task we move towards the array  String [] names = new String[150]  As per above line we can access 150 student data with the help of array easily  Limitation of an array===============================  1.Size is fixed  2.Stroes only homogeneous data  To overcome above we move towards Collection  Collection is an ***INTERFACE***  Why we need Collection===  1.We need collection for grouping number of elements or data on the single entity.  2.To overcome the limitation of an array  In version 1.5 2 collection was introduced  1.vector  2.Hash table  ***Collection category***—does not derived from collection interface.  1.List – list of things  2.set – care about uniqueness  3.map – stores key, value pairs.  ***Collection methods --*** It provides method without body implementation.  NOTE ==  Method present in collection interface ,interface should be implemented.  1.Insert/add – add(object)  2.read – iterator(object) – we cannot read single value we have to read complete value  3.Remove – remove(object)  4.search – search(object) – It has by default search. Does not required binary and linear searchs  5.size – size(object)  6.empty – IsEmpty() – Ans is in Boolean value i.e true or false  **NOTE ==**  collection only support object type n does not support primitive type bcz, we have to create object and does not work on primitive type. |

#### 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:
* Interfaces and its implementations, i.e., classes
* 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.



### 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:

## 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 properties and behaviors is acquire by the child .
* It contains only one abstract method. i.e.,
* 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. (Underlying Data structure Array or uses array to stored data)
* The ArrayList class **maintains** the **insertion order**(FIFO) and is **non-synchronized.**
* The elements stored in the ArrayList class can be **randomly accessed.**
* It is an growable array i.e It can grow until the JVM provides the memory
* It allows duplicate value
* It allows null to store
* It does not allow any sorting technique.
* Stores heterogeneous data or elements

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 list interface.
* It uses a doubly linked list internally to store the elements.( Underlying data structure is doubly linked list)
* 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.
* It allows null value
* sorting is not possible
* It does not use array to stored elements ,It uses list of nodes to stored values
* value not required to stored continuously

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

Ravikam

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 (LIFO)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 (FIFO)order.
* It can be defined as an ordered list that is used to hold the elements which are about to be processed.
* There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.
* Queue interface can be instantiated as:
* Queue<String> q1 = **new** PriorityQueue();
* Queue<String> q2 = **new** ArrayDeque();
* 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:
* 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 cares about uniqueness
* 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.

1. It is not using array to store
2. It uses hast table r underlying DS is hashtable(Calculate the hashcode and then stored the value)
3. It is not index based
4. It allows only null value
5. It does not follow any insertion order
6. It does not allow any sorting

**Hashset Methods===========**

* add(object)
* get(index)
* remove(object)
* iterator
* size()
* IsEmpty()
* Contains()

**NOTE:** hashcode generation===Integer value generator based on the local address where the value is saved. It is default implementation.

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.
* It does not allowed duplicates.
* Only one null value is allowed
* Underlying DS is doubly LL & hash Table
* Sorting is not possible
* It is not index based.

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(natural sorting).
* No duplicates allowed
* Insertion order I not maintaind
* No null values are allowed
* Underlying DS is balanced tree
* It implements sorted set and navigable set interface along set interfaces

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 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.

|  |
| --- |
| *1.List*  It cares about index or it is an index based collection  List is an interface i.e interface list implements collection means method present in collection should be implemented by List.  Method of List ===  1.add(index, object)  2.remove(index)  3.set (index, object) -update  List L = new List(); //as we know that ,it is not possible bcz only class object is made so we have need to have an implementation of list .  **We can implements list by two ways===**  1.ArrayList implementation  2.LinkedList implementation |

### 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.

### 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.

### 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.

### 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**.

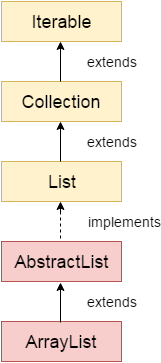
### How to Sort List

* 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.

## Java ListIterator Interface

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

# *Java ArrayList*



* Java **ArrayList** class uses a dynamic [*array*](https://www.javatpoint.com/array-in-java) for storing the elements.
* It is like an array, but there is no size limit.
* We can add or remove elements anytime.
* So, it is much more flexible than the traditional array.
* It is found in the java.util package. It is like the Vector in C++.
* The ArrayList in Java can have the duplicate elements also.
* It implements the List interface so we can use all the methods of List interface here.
* The ArrayList maintains the insertion order internally.
* It inherits the AbstractList class and implements [List interface](https://www.javatpoint.com/java-list).

The important points about Java ArrayList class are:

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non [synchronized](https://www.javatpoint.com/synchronization-in-java).
* Java ArrayList allows random access because array works at the index basis.
* In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

|  |
| --- |
| 1.ArrayList implementation=======================  Class ArrayList implements List  {  //need to provide implemtation for all the methods in the list interface & collection interface  //As we knew that list implements collection so we have to implements or iuse all methods of collection  }  **Features of ArrayList=======**  1.It is an growable array i.e It can grow until the JVM provides the memory  2.Underlying Data structure Array or uses array to stored data  3.It allows duplicate value  4.Insertion order is maintained(FIFO) or it is an order collection  5.It allows null to store  6.It does not allow any sorting technique.  7.Stores heterogeneous data or elements |

### Hierarchy of ArrayList class

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

### ArrayList class declaration

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

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

### Constructors of ArrayList

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

### 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.

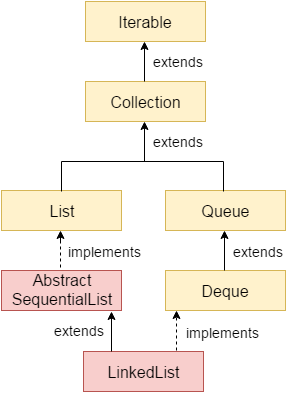
Programs=================

|  |
| --- |
| ArrayListDemo  package collection;  import java.util.\*;  public class ArrayListDemo  {  public static void main(String args[])  {  ArrayList<String> list = new ArrayList<>(); //Specified for String Element or Data    //Add  list.add("Apple");  list.add("Orange");  list.add("Banana");  list.add("Papaya");    //Read Approach 1 -- Do not use ths type of Approach for Debugging it is used  System.out.println(list);    System.out.println();    //Read Approach 2  for(String s:list)  {  System.out.println(s);  }    System.out.println();    //Read Approach 3 - Iterator  Iterator<String> iterator = list.iterator(); //Iterator Similar to cursor or pointer  while(iterator.hasNext())  {  String a = iterator.next();  System.out.println(a);  }  }  } |

|  |
| --- |
| ArrayListDemo1 – utilizing all the methods in this pgm  package collection;  import java.util.\*;  // Size not Fixed  // Duplicate Allowedd  // Insertion Ordder Maintained FIFO  // N number of null value can be Inserted  // No sorting is possible  class ArrayListDemo2  {  public static void main(String args[])  {  ArrayList<String> list = new ArrayList<>(); //Specified for String Element or Data    //Add or Insert  list.add("Apple");  list.add("Orange");  list.add("Banana");  list.add("Apple");  list.add("Orange");  list.add("Banana");  list.add("Papaya");  list.add(null);    System.out.println("Size of the Array :"+list.size()); //Array Size  System.out.println("is Empty :"+list.isEmpty()); //Array is Empty or Not Answer is in Truue or False  System.out.println("Contains Apple :"+list.contains("Apple"));//Check Apple present or not Answer is in Truue or False  System.out.println("Contains apple :"+list.contains("apple")); //Case Sensitive answer will be False    //Remove using Index  String elt = list.remove(0);  System.out.println("Removed :"+elt);    //Check Removed or not  boolean flag =list.remove("Orange");  System.out.println("is orange Removed :"+flag);    flag =list.remove("Cherry");  System.out.println("is Cherry Removed :"+flag);      //Other Methods to Add  System.out.println("Size of the Array :"+list.size());  System.out.println("Before adding cherry");  list.add(0,"Cherry"); //It will store to 0 index rest all values increment by 1  //By putting index we can add anywhere  System.out.println("Size of the Array :"+list.size());  System.out.println(list);  list.set(0,"No Fruit"); //Updating the 0 index value with no fruit or replacing  System.out.println(list);    //Read Approach 3 - Iterator  Iterator<String> iterator = list.iterator(); //Iterator Similar to cursor or pointer  while(iterator.hasNext())  {  String a = iterator.next();  System.out.println(a);  }  }  } |

|  |
| --- |
| 3.ArrayListHetregenous  package collection;  //ArrayListHetregenouspackage collection;  import java.util.ArrayList;  // Size not Fixed  // Duplicate Allowedd  // Insertion Ordder Maintained  // N number of null value can be Inserted  // No sorting is possible  public class ArrayListHetregenous  {  public static void main(String args[])  {  ArrayList list = new ArrayList<>();    ArrayList<Object> list2 = new ArrayList<>();  list.add("one");  list.add(1001);  list.add(true);  list.add(3.15);  list.add(new Student(122,"Swapnali"));    int index = 4;    if(list.get(index)instanceof String)  {  String s = (String) list.get(index);  System.out.println("String : "+s);  }else if(list.get(index)instanceof Integer)  {  Integer i = (Integer)list.get(index);  System.out.println("Integer : "+i);  }else if(list.get(index)instanceof Student)  {  Student ss = (Student)list.get(index);  System.out.println("Student : "+ss);  }  System.out.println(list);  }  } |

***Java LinkedList class***



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

The important points about Java LinkedList are:

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

Hierarchy of LinkedList class

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

Doubly Linked List

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



|  |
| --- |
| **Linked List implementation======================**  1.It allows duplicate value  2.Underlying data structure is doubly linked list  3.It allows null vlue  4.Insertion order is maintained  5.sorting is not possible  6.Each element is doubly Linked to each other  7.It does not use array to stored elements ,It uses list of nodes to stored values  8.value not required to stored continuously  Linked list Method====  1.addFirst()  2.addLast()  3.removeFirst()  4removeLast()  5getFirst()  6.getLast()  7.get(index)  8.add(object) – adding the value to the end of the list.  9.remove() – Remove the value from the front |

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. |

Programs=========================================

|  |
| --- |
| Dobly LL=====  package collection;  import java.util.LinkedList;  public class LinkedListDemo  {  public static void main(String args[])  {  LinkedList<Integer>list = new LinkedList<>();  list.add(10);  list.addFirst(20);  list.add(30);  list.add(15);  list.add(40);    System.out.println(list);  System.out.println("Get first : + "+list.getFirst());  System.out.println("Get last : + "+list.getLast());      list.removeFirst();  list.removeLast();  }    } |

# Difference between ArrayList and LinkedList

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

However, there are many differences between 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 bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3) An ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

***Diiference between array List & LinkedList***

1 Arraybased DoublyLinkedListBased

2.Stiored in continuous memory does not Stored in continuous memory

3.Adding and reading is easy Adding and reading is costlier

4.Adding in middle is costlier(bcz we need to shify rest of thr array) Insertion in middle is easy.

Java Queue Interface

* Java Queue interface orders the element in FIFO(First In First Out) manner.
* In FIFO, first element is removed first and last element is removed at last.

Queue Interface declaration

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

Methods of Java Queue Interface

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

PriorityQueue class

* The PriorityQueue class provides the facility of using queue.
* But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.

### Java PriorityQueue Example

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

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection12)

Output:head:Amit

head:Amit

iterating the queue elements:

Amit

Jai

Karan

Vijay

Rahul

after removing two elements:

Karan

Rahul

Vijay

Java Deque Interface

* Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**

Deque Interface declaration

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

Methods of Java Deque Interface

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

ArrayDeque class

* The ArrayDeque class provides the facility of using deque and resizable-array.
* It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

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

ArrayDeque Hierarchy

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

ArrayDeque class declaration

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

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

Java ArrayDeque Example

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

Output:

Ravi

Vijay

Ajay

# Java 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:

* 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. |

## 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.

# *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. |

### 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**](http://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.

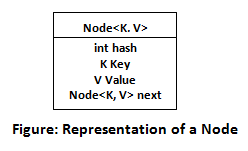
***Working of HashMap in Java***

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

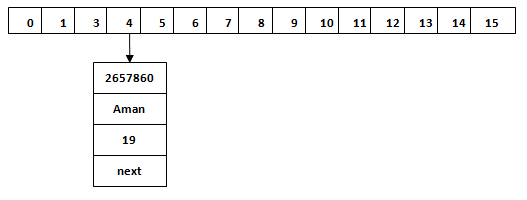
* Index minimizes the size of the array. The Formula for calculating the index is:

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.

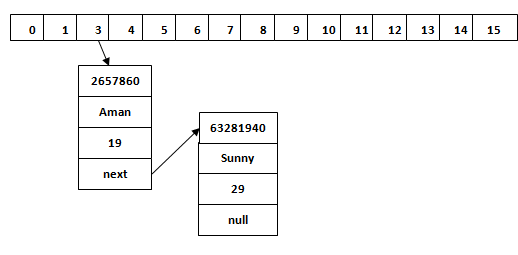


Hash Collision

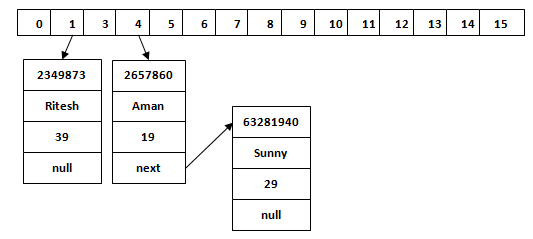
* This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

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

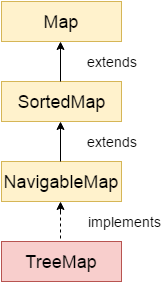
1 Details:

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.

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

TreeMap class Parameters

* Let's see the Parameters for java.util.TreeMap class.
* **K**: It is the type of keys maintained by this map.
* **V**: It is the type of mapped values.

Constructors of Java TreeMap class

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

Java TreeMap Example

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

Output:100 Amit

101 Vijay

102 Ravi

103 Rahul

Java TreeMap Example: remove()

1. **import** java.util.\*;
2. **public** **class** TreeMap2 {
3. **public** **static** **void** main(String args[]) {
4. TreeMap<Integer,String> map=**new** TreeMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(102,"Ravi");
7. map.put(101,"Vijay");
8. map.put(103,"Rahul");
9. System.out.println("Before invoking remove() method");
10. **for**(Map.Entry m:map.entrySet())
11. {
12. System.out.println(m.getKey()+" "+m.getValue());
13. }
14. map.remove(102);
15. System.out.println("After invoking remove() method");
16. **for**(Map.Entry m:map.entrySet())
17. {
18. System.out.println(m.getKey()+" "+m.getValue());
19. }
20. }
21. }

Output:

Before invoking remove() method

100 Amit

101 Vijay

102 Ravi

103 Rahul

After invoking remove() method

100 Amit

101 Vijay

103 Rahul

Java TreeMap Example: NavigableMap

1. **import** java.util.\*;
2. **class** TreeMap3{
3. **public** **static** **void** main(String args[]){
4. NavigableMap<Integer,String> map=**new** TreeMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(102,"Ravi");
7. map.put(101,"Vijay");
8. map.put(103,"Rahul");
9. //Maintains descending order
10. System.out.println("descendingMap: "+map.descendingMap());
11. //Returns key-value pairs whose keys are less than or equal to the specified key.
12. System.out.println("headMap: "+map.headMap(102,**true**));
13. //Returns key-value pairs whose keys are greater than or equal to the specified key.
14. System.out.println("tailMap: "+map.tailMap(102,**true**));
15. //Returns key-value pairs exists in between the specified key.
16. System.out.println("subMap: "+map.subMap(100, **false**, 102, **true**));
17. }
18. }

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

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

tailMap: {102=Ravi, 103=Rahul}

subMap: {101=Vijay, 102=Ravi}

Java TreeMap Example: SortedMap

1. **import** java.util.\*;
2. **class** TreeMap4{
3. **public** **static** **void** main(String args[]){
4. SortedMap<Integer,String> map=**new** TreeMap<Integer,String>();
5. map.put(100,"Amit");
6. map.put(102,"Ravi");
7. map.put(101,"Vijay");
8. map.put(103,"Rahul");
9. //Returns key-value pairs whose keys are less than the specified key.
10. System.out.println("headMap: "+map.headMap(102));
11. //Returns key-value pairs whose keys are greater than or equal to the specified key.
12. System.out.println("tailMap: "+map.tailMap(102));
13. //Returns key-value pairs exists in between the specified key.
14. System.out.println("subMap: "+map.subMap(100, 102));
15. }
16. }

headMap: {100=Amit, 101=Vijay}

tailMap: {102=Ravi, 103=Rahul}

subMap: {100=Amit, 101=Vijay}

What is difference between HashMap and TreeMap?

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

Java TreeMap Example: Book

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

Output:

1 Details:

101 Let us C Yashwant Kanetkar BPB 8

2 Details:

102 Data Communications & Networking Forouzan Mc Graw Hill 4

3 Details:

103 Operating System Galvin Wiley 6

# *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. |

### 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**](http://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 Collections class**

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

The important points about Java Collections class are:

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

**Sorting in Collection**

We can sort the elements of:

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

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

Method of Collections class for sorting List elements

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

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.

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

Properties class in Java

* The **properties** object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.
* It can be used to get property value based on the property key. The Properties class provides methods to get data from the properties file and store data into the properties file. Moreover, it can be used to get the properties of a system.

An Advantage of the properties file

* **Recompilation is not required if the information is changed from a properties file:** If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.

Example of Properties class to get information from the properties file

* To get information from the properties file, create the properties file first.

**db.properties**

1. user=system
2. password=oracle

Example of Properties class to get all the system properties

* By System.getProperties() method we can get all the properties of the system. Let's create the class that gets information from the system properties.

**Test.java**

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** Test {
4. **public** **static** **void** main(String[] args)**throws** Exception{
6. Properties p=System.getProperties();
7. Set set=p.entrySet();
9. Iterator itr=set.iterator();
10. **while**(itr.hasNext()){
11. Map.Entry entry=(Map.Entry)itr.next();
12. System.out.println(entry.getKey()+" = "+entry.getValue());
13. }
15. }
16. }

Output:

java.runtime.name = Java(TM) SE Runtime Environment

sun.boot.library.path = C:\Program Files\Java\jdk1.7.0\_01\jre\bin

java.vm.version = 21.1-b02

java.vm.vendor = Oracle Corporation

java.vendor.url = http://java.oracle.com/

path.separator = ;

java.vm.name = Java HotSpot(TM) Client VM

file.encoding.pkg = sun.io

user.country = US

user.script =

sun.java.launcher = SUN\_STANDARD

...........

Example of Properties class to create the properties file

Now let's write the code to create the properties file.

**Test.java**

1. **import** java.util.\*;
2. **import** java.io.\*;
3. **public** **class** Test {
4. **public** **static** **void** main(String[] args)**throws** Exception{
6. Properties p=**new** Properties();
7. p.setProperty("name","Sonoo Jaiswal");
8. p.setProperty("email","sonoojaiswal@javatpoint.com");
10. p.store(**new** FileWriter("info.properties"),"Javatpoint Properties Example");
12. }
13. }

Let's see the generated properties file.

**info.properties**

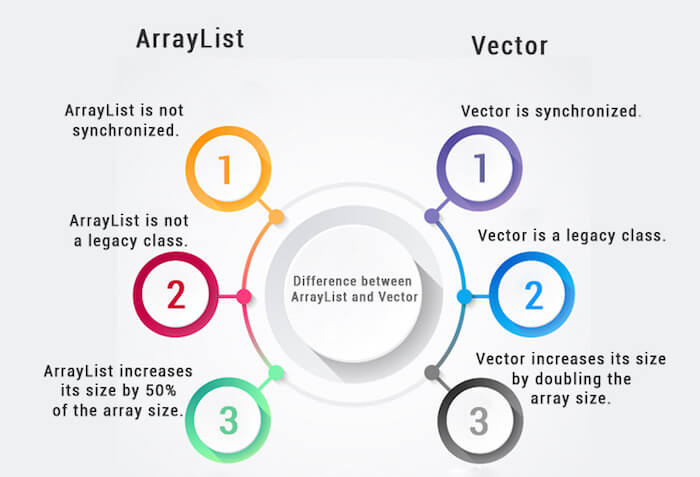
1. #Javatpoint Properties Example
2. #Thu Oct 03 22:35:53 IST 2013
3. email=sonoojaiswal@javatpoint.com
4. name=Sonoo Jaiswal

# Difference between ArrayList and Vector

ArrayList and Vector both implements List interface and maintains insertion order.

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

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity. | Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object. |
| 5) ArrayList uses the **Iterator** interface to traverse the elements. | A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements. |



### Example of Java ArrayList

Let's see a simple example where we are using ArrayList to store and traverse the elements.

1. **import** java.util.\*;
2. **class** TestArrayList21{
3. **public** **static** **void** main(String args[]){
5. List<String> al=**new** ArrayList<String>();//creating arraylist
6. al.add("Sonoo");//adding object in arraylist
7. al.add("Michael");
8. al.add("James");
9. al.add("Andy");
10. //traversing elements using Iterator
11. Iterator itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestArrayList21)

Output:

Sonoo

Michael

James

Andy

### Example of Java Vector

Let's see a simple example of a Java Vector class that uses the Enumeration interface.

1. **import** java.util.\*;
2. **class** TestVector1{
3. **public** **static** **void** main(String args[]){
4. Vector<String> v=**new** Vector<String>();//creating vector
5. v.add("umesh");//method of Collection
6. v.addElement("irfan");//method of Vector
7. v.addElement("kumar");
8. //traversing elements using Enumeration
9. Enumeration e=v.elements();
10. **while**(e.hasMoreElements()){
11. System.out.println(e.nextElement());
12. }
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestVector1)

Output:

umesh

irfan

kumar

***Java Vector***

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

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

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

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

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

Java Vector class Declaration

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

|  |  |  |
| --- | --- | --- |
| Properties | Arraylist | LinkedList |
| List | Arraylist is a class | LinkedList is a class |
|  | Arraylist extends List interface | LinkedList extends List interface |
| Duplicate | Allow | Allow |
| Insertion order | Maintained | Maintained |
| Null | Allow | Allow |
| Store | Dynamic array | Doubly-Linked list |
|  | Non-synchronized | Non-synchronized |
| Manipulation | Slow(shifting required) | Fast(no shifting required) |
| Better for | **better for storing and accessing** data. | **better for** Manipulation |

### 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. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **HashMap** | | **Hashtable** | | |
| 1) HashMap is **non synchronized**. | | Hashtable is **synchronized**. |
| 2) HashMap **allows one null key and multiple null values**. | | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is **fast**. | | Hashtable is **slow**. |
| **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. | | |