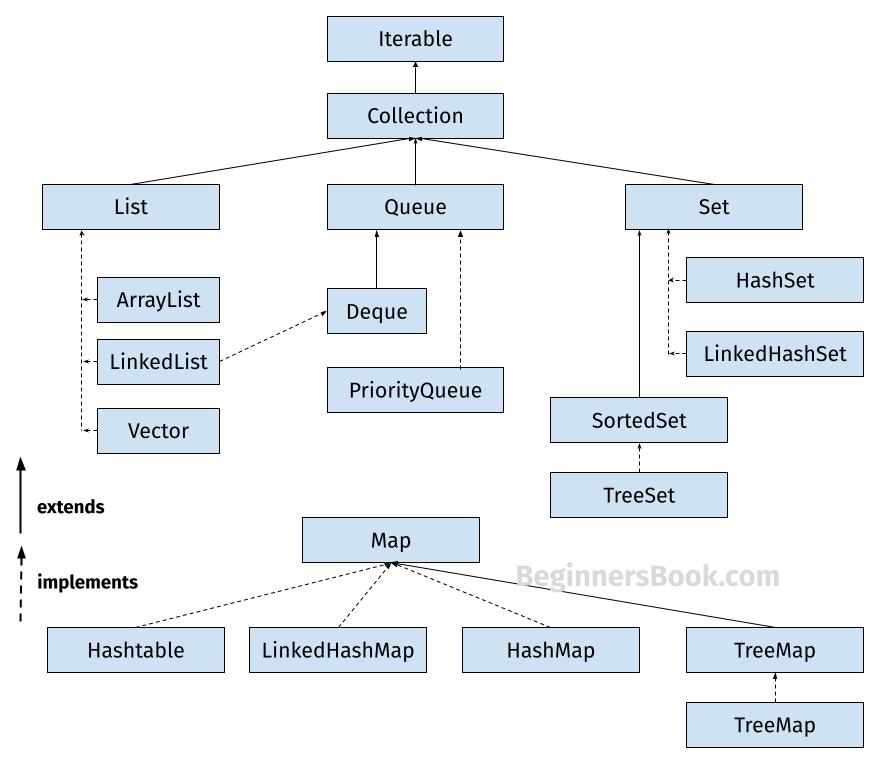
# Collections in Java

The **Java Collections Framework** is a collection of interfaces and classes, which helps in storing and processing the data efficiently

## Collections Framework hierarchy



**1. List**

A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements. Elements can be inserted or accessed by their position in the list, using a zero-based index. The classes that implements List interface are:

* ArrayList
* LinkedList
* Vector
* Stack

**1.1 ArrayList**

ArrayList is a popular alternative of [arrays in Java](https://beginnersbook.com/2013/05/java-arrays/). It is based on an Array **data structure**. ArrayList is a resizable-array implementation of the List interface. It implements all optional list operations.

import java.util.\*;

class JavaExample{

public static void main(String args[]){

//creating ArrayList of string type

ArrayList<String> arrList=new ArrayList<>();

//adding few elements

arrList.add("Cricket"); //list: ["Cricket"]

arrList.add("Hockey"); //list: ["Cricket", "Hockey"]

//inserting element at first position, index 0

//represents first element because ArrayList is based

//on zero based indexing system

arrList.add(0, "BasketBall"); //list: ["BasketBall", "Cricket", "Hockey"]

System.out.println("ArrayList Elements: ");

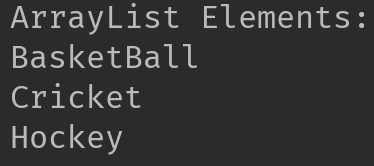
//Traversing ArrayList using enhanced for loop

for(String str:arrList)

System.out.println(str);

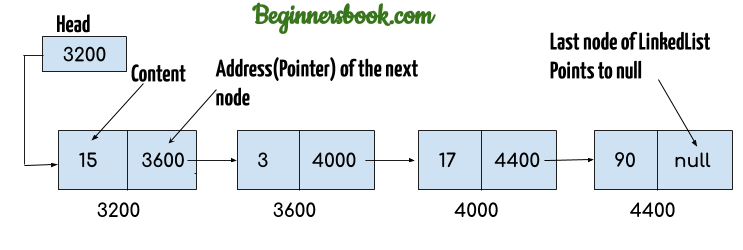
}

}

**Output:**  


.

**1.2 LinkedList**

LinkedList is a linear data structure. However LinkedList elements are not stored in contiguous locations like arrays, they are linked with each other using pointers. Each element of the LinkedList has the reference(address/pointer) to the next element of the LinkedList.  


import java.util.\*;

public class JavaExample{

public static void main(String args[]){

LinkedList<String> linkList=new LinkedList<>();

linkList.add("Apple"); //["Apple"]

linkList.add("Orange"); //["Apple", "Orange"]

//inserting element at first position

linkList.add(0, "Banana"); ////["Banana", "Apple", "Orange"]

System.out.println("LinkedList elements: ");

//iterating LinkedList using iterator

Iterator<String> it=linkList.iterator();

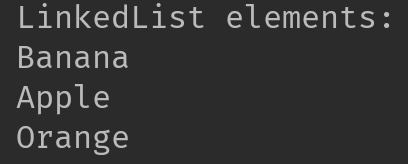
while(it.hasNext()){

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

}

}

}

**Output:**  


**1.3 Vector**

Here is the list of all the tutorials published on the Vector.

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

Vector<String> v=new Vector<>();

v.add("item1"); //["item1"]

v.add("item2"); //["item1", "item2"]

v.add("item3"); //["item1", "item2", "item3"]

//removing an element

v.remove("item2"); //["item1", "item3"]

System.out.println("Vector Elements: ");

//iterating Vector using iterator

Iterator<String> it=v.iterator();

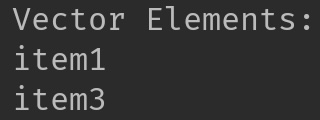
while(it.hasNext()){

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

}

}

}

**Output:**  
  
Refer [this article](https://beginnersbook.com/2013/12/vector-in-java/) for more **guides on Vector**.

**1.4 Stack**

Stack class extends Vector class, which means it is a subclass of Vector. Stack works on the concept of Last In First Out (LIFO). The elements are inserted using push() method at the end of the stack, the pop() method removes the element which was inserted last in the Stack.

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

Stack<String> stack = new Stack<>();

//push() method adds the element in the stack

//and pop() method removes the element from the stack

stack.push("Chaitanya"); //["Chaitanya"]

stack.push("Ajeet"); //["Chaitanya", Ajeet]

stack.push("Hari"); //["Chaitanya", "Ajeet", "Hari"]

stack.pop(); //removes the last element

stack.push("Steve"); //["Chaitanya", "Ajeet", "Steve"]

stack.push("Carl"); //["Chaitanya", "Ajeet", "Steve", "Carl"]

stack.pop(); //removes the last element

System.out.println("Stack elements: ");

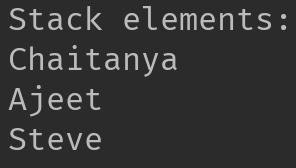
for(String str: stack){

System.out.println(str);

}

}

}

**Output:**  


**2. Set**

A Set is a Collection that cannot contain duplicate elements. There are three main implementations of Set interface: HashSet, TreeSet, and LinkedHashSet.

**2.1 HashSet**

[HashSet](https://beginnersbook.com/2013/12/hashset-class-in-java-with-example/) which stores its elements in a hash table, is the best-performing implementation. HashSet allows only unique elements. It doesn’t maintain the insertion order which means element inserted last can appear at first when traversing the HashSet.

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

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

set.add("Paul");

set.add("Ram");

set.add("Aaron");

set.add("Leo");

set.add("Becky");

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

while(it.hasNext()){

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

}

}

}

**Output:**

Aaron

Leo

Paul

Ram

Becky

**2.2 LinkedHashSet**

Unlike HashSet, the [LinkedHashSet](https://beginnersbook.com/2013/12/linkedhashset-class-in-java-with-example/) maintains insertion order.

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

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

set.add("Paul");

set.add("Ram");

set.add("Aaron");

set.add("Leo");

set.add("Becky");

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

while(it.hasNext()){

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

}

}

}

**Output:**

Paul

Ram

Aaron

Leo

Becky

**2.3 TreeSet**

[TreeSet](https://beginnersbook.com/2013/12/treeset-class-in-java-with-example/) stores elements in a red-black tree. It is substantially slower than HashSet. TreeSet class implements SortedSet interface, which allows TreeSet to order its elements based on their values, which means TreeSet elements are sorted in ascending order.

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

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

set.add("Paul");

set.add("Ram");

set.add("Aaron");

set.add("Leo");

set.add("Becky");

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

while(it.hasNext()){

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

}

}

}

**Output:**

Aaron

Becky

Leo

Paul

Ram

**3. Map**

A Map is an object that maps keys to values. A map cannot contain duplicate keys. There are three main implementations of Map interfaces: HashMap, TreeMap, and LinkedHashMap.

**3.1 HashMap**

**HashMap:** HashMap is like HashSet, it doesn’t maintain insertion order and doesn’t sort the elements in any order.

public class JavaExample{

public static void main(String args[]){

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

//key and value pairs

hmap.put(101, "Chaitanya");

hmap.put(105, "Derick");

hmap.put(111, "Logan");

hmap.put(120, "Paul");

//print HashMap elements

Set set = hmap.entrySet();

Iterator iterator = set.iterator();

while(iterator.hasNext()) {

Map.Entry m = (Map.Entry)iterator.next();

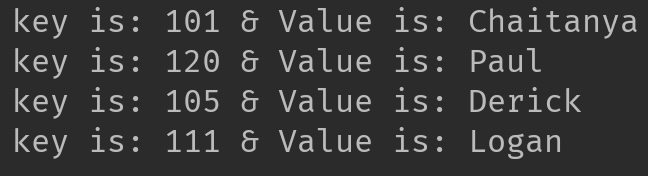
System.out.print("key is: "+ m.getKey() + " & Value is: ");

System.out.println(m.getValue());

}

}

}

**Output:**  


**3.2 TreeMap**

**TreeMap:** It stores its elements in a red-black tree. The elements of TreeMap are sorted in ascending order. It is substantially slower than HashMap.This is the same example that we have seen above in HashMap. Here, elements are sorted based on keys.

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

TreeMap<Integer, String> hmap = new TreeMap<>();

//key and value pairs

hmap.put(101, "Chaitanya");

hmap.put(105, "Derick");

hmap.put(111, "Logan");

hmap.put(120, "Paul");

//print HashMap elements

Set set = hmap.entrySet();

Iterator iterator = set.iterator();

while(iterator.hasNext()) {

Map.Entry m = (Map.Entry)iterator.next();

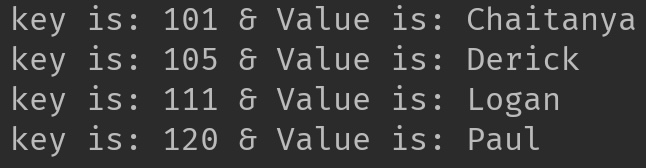
System.out.print("key is: "+ m.getKey() + " & Value is: ");

System.out.println(m.getValue());

}

}

}

**Output:**  


**3.3 LinkedHashMap**

**LinkedHashMap:** It maintains insertion order. Refer [this guide](https://beginnersbook.com/2013/12/linkedhashmap-in-java/), to learn LinkedHashMap in detail. As you can see: In the following example, the key & value pairs maintained the insertion order.

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

LinkedHashMap<Integer, String> hmap = new LinkedHashMap<>();

//key and value pairs

hmap.put(100, "Chaitanya");

hmap.put(120, "Paul");

hmap.put(105, "Derick");

hmap.put(111, "Logan");

//print LinkedHashMap elements

Set set = hmap.entrySet();

Iterator iterator = set.iterator();

while(iterator.hasNext()) {

Map.Entry m = (Map.Entry)iterator.next();

System.out.print("key is: "+ m.getKey() + " & Value is: ");

System.out.println(m.getValue());

}

}

}

**Output:**

key is: 100 & Value is: Chaitanya

key is: 120 & Value is: Paul

key is: 105 & Value is: Derick

key is: 111 & Value is: Logan

**ArrayList in Java With Examples**

**Arraylist** class implements List interface and it is based on an Array data structure. It is widely used because of the functionality and flexibility it offers. **ArrayList in Java**, is a resizable-array implementation of the List interface

**Array vs arraylist in java**

The main **difference between array and arraylist** is that arraylist can grow and shrink dynamically while an array cannot.

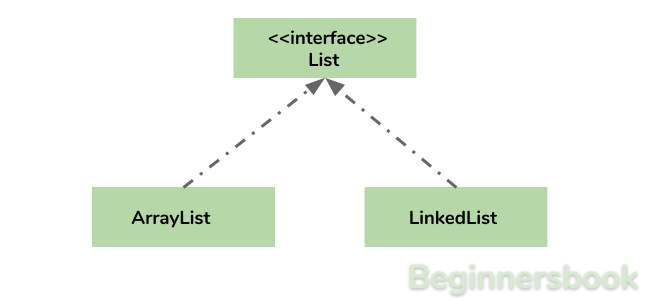
An array has a **fixed length** so if it is full you cannot add any more elements to it. Similarly, if number of elements are removed from ArrayList, the memory consumption remains same as it doesn’t shrink.

On the other hand, **ArrayList can dynamically grow and shrink** after addition and removal of elements. ArrayList class has several useful methods that can make our task easy.

**ArrayList in Java**

* ArrayList can grow and shrink automatically based on the addition and removal of elements.
* ArrayList can contain duplicate elements
* ArrayList maintains the insertion order, which means the elements appear in the same order in which they are inserted.
* ArrayList is non synchronized. However you can make it [synchronized](https://beginnersbook.com/2013/12/how-to-synchronize-arraylist-in-java-with-example/).

**Hierarchy of ArrayList class in Java**

ArrayList class implements List interface and List interface extends Collection interface.  


**Arraylist in Java declaration**

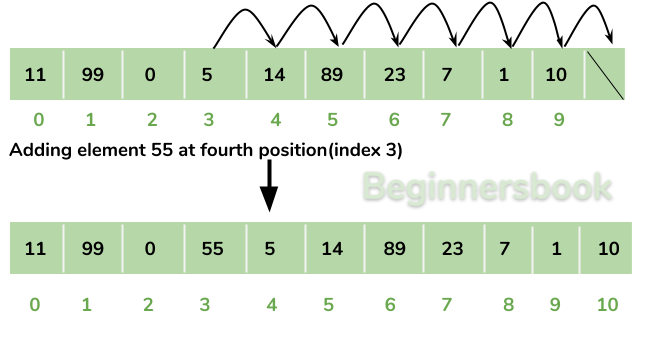
This is how you can declare an ArrayList of String type:

ArrayList<String> list=new ArrayList<>();

This is how you can declare an ArrayList of Integer type:

ArrayList<Integer> list=new ArrayList<>();

**Adding elements to Arraylist in java**

**Adding Element in ArrayList at specified position:**  
  
You can add elements to an ArrayList by using [add() method](https://beginnersbook.com/2013/12/java-arraylist-add-method-example/). This method has couple of variations, which you can use based on the requirement.

**For example:** If you want to add the element at the end of the List then you can simply call the add() method like this:

arrList.add("Steve"); //This will add "Steve" at the end of List

To add the element at the specified location in ArrayList, you can specify the index in the add() method like this:

arrList.add(3, "Steve"); //This will add "Steve" at the fourth position

Lets write the complete code:

import java.util.\*;

class JavaExample{

public static void main(String args[]){

ArrayList<String> arrList=new ArrayList<String>();

arrList.add("Steve");

arrList.add("Tim");

arrList.add("Lucy");

arrList.add("Pat");

arrList.add("Angela");

arrList.add("Tom");

//displaying elements

System.out.println(arrList);

//Adding "Steve" at the fourth position

arrList.add(3, "Steve");

//displaying elements

System.out.println(arrList);

}

}

**Output:**

[Steve, Tim, Lucy, Pat, Angela, Tom]

[Steve, Tim, Lucy, Steve, Pat, Angela, Tom]

**Note:** Since the index starts with 0, index 3 would represent fourth position not 3.

**Change an element in ArrayList**

You can use the **set method** to change an element in ArrayList. You need to provide the **index** and **new element**, this method then updates the element present at the **given index** with the **new given element**.

In the following example, we have given the index as 0 and new element as “Lucy” in the set() method. The method updated the element present at the index 0 (“Jim”) with the new String element “Lucy”.

import java.util.ArrayList;

public class JavaExample {

public static void main(String[] args) {

ArrayList<String> names = new ArrayList<String>();

names.add("Jim");

names.add("Jack");

names.add("Ajeet");

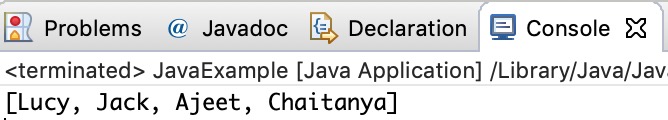
names.add("Chaitanya");

names.set(0, "Lucy");

System.out.println(names);

}

}

**Output:**  


**How to remove element from Arraylist in Java?**

**Removing Element from ArrayList:**  
  
You can use **remove() method** to remove elements from an ArrayList. Similar to add() method, this method also has couple of variations.

**For example:**

import java.util.\*;

class JavaExample{

public static void main(String args[]){

ArrayList<String> alist=new ArrayList<String>();

alist.add("Steve");

alist.add("Tim");

alist.add("Lucy");

alist.add("Pat");

alist.add("Angela");

alist.add("Tom");

//displaying elements

System.out.println(alist);

//Removing "Steve" and "Angela"

alist.remove("Steve");

alist.remove("Angela");

//displaying elements

System.out.println(alist);

//Removing 3rd element

alist.remove(2);

//displaying elements

System.out.println(alist);

}

}

**Output:**

[Steve, Tim, Lucy, Pat, Angela, Tom]

[Tim, Lucy, Pat, Tom]

[Tim, Lucy, Tom]

**Iterating ArrayList**

Here, we are using enhanced for loop to iterate ArrayList elements. This one of the best ways to iterate an ArrayList of string type.

import java.util.\*;

class JavaExample{

public static void main(String args[]){

ArrayList<String> alist=new ArrayList<String>();

alist.add("Gregor Clegane");

alist.add("Khal Drogo");

alist.add("Cersei Lannister");

alist.add("Sandor Clegane");

alist.add("Tyrion Lannister");

//iterating ArrayList

for(String str:alist)

System.out.println(str);

}

}

**Output:**

Gregor Clegane

Khal Drogo

Cersei Lannister

Sandor Clegane

Tyrion Lannister

**ArrayList Size**

We can use size() method of ArrayList to find the **number of elements in an ArrayList**.

import java.util.ArrayList;

public class JavaExample {

public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(1);

numbers.add(7);

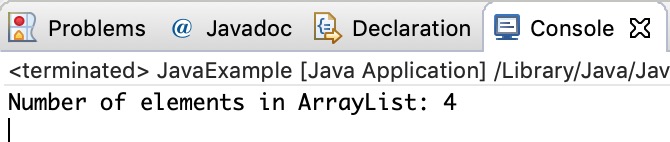
numbers.add(5);

numbers.add(6);

System.out.println("Number of elements in ArrayList: "+numbers.size());

}

}

**Output:**  


**Sort ArrayList**

You can use the **sort() method** of the **Collections utility class**to [sort an ArrayList](https://beginnersbook.com/2013/12/how-to-sort-arraylist-in-java/). This class is is a part of **java.util** package. In the following example we are sorting a list of String type **alphabetically.** This method also works on **numeric lists** (such as Integer type ArrayList).

import java.util.ArrayList;

import java.util.Collections;

public class JavaExample {

public static void main(String[] args) {

ArrayList<String> fruits = new ArrayList<String>();

fruits.add("Orange");

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Pineapple");

Collections.sort(fruits);

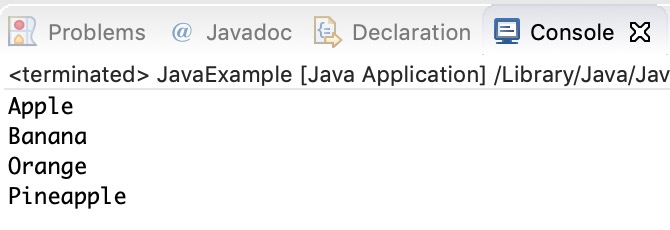
for (String str : fruits) {

System.out.println(str);

}

}

}

**Output:**  


**ArrayList Example in Java**

This example demonstrates, how to **create**, **initialize**, **add** and **remove** elements from ArrayList. In this example we have an ArrayList of “String” type. We are adding 5 String element in the ArrayList using the method add(String E). This method adds the element at the end of the ArrayList.

We are then **adding** two more elements in the ArrayList using method add(int index, String E). This method adds the specified element at the specified index, index 0 indicates first position, 1 indicates second position and so on.

We are then **removing** the elements “Chaitanya” and “Harry” from the ArrayList. We again removed the second element of the ArrayList using method remove(int index).

import java.util.\*;

public class JavaExample {

public static void main(String args[]) {

/\* Creating ArrayList of type "String" which means

\* we can only add "String" elements

\*/

ArrayList<String> obj = new ArrayList<String>();

/\*This is how we add elements to an ArrayList\*/

obj.add("Ajeet");

obj.add("Harry");

obj.add("Chaitanya");

obj.add("Steve");

obj.add("Anuj");

// Displaying elements

System.out.println("Original ArrayList:");

for(String str:obj)

System.out.println(str);

/\* Add element at the given index

\* obj.add(0, "Rahul") - Adding element "Rahul" at first position

\* obj.add(1, "Justin") - Adding element "Justin" at second position

\*/

obj.add(0, "Rahul");

obj.add(1, "Justin");

// Displaying elements

System.out.println("ArrayList after add operation:");

for(String str:obj)

System.out.println(str);

//Remove elements from ArrayList like this

obj.remove("Chaitanya"); //Removes "Chaitanya" from ArrayList

obj.remove("Harry"); //Removes "Harry" from ArrayList

// Displaying elements

System.out.println("ArrayList after remove operation:");

for(String str:obj)

System.out.println(str);

//Remove element from the specified index

obj.remove(1); //Removes Second element from the List

// Displaying elements

System.out.println("Final ArrayList:");

for(String str:obj)

System.out.println(str);

}

}

**Output:**

Original ArrayList:

Ajeet

Harry

Chaitanya

Steve

Anuj

ArrayList after add operation:

Rahul

Justin

Ajeet

Harry

Chaitanya

Steve

Anuj

ArrayList after remove operation:

Rahul

Justin

Ajeet

Steve

Anuj

Final ArrayList:

Rahul

Ajeet

Steve

Anuj

**All methods of Arraylist in Java**

In the above examples, we have used methods such as add() and remove(). However there are number of other useful methods available in ArrayList class.

1) [**add**](https://beginnersbook.com/2013/12/java-arraylist-add-method-example/)**( Object o)**: This method adds an object o at the end of the arraylist.

obj.add("hello");

This statement would add a string hello in the arraylist at last position.

2) [**add**](https://beginnersbook.com/2013/12/java-arraylist-addint-index-e-element-example/)**(int index, Object o)**: It adds the object o at the specified index in the ArrayList.

obj.add(2, "bye");

It will add the string “bye” at the 2nd index (third element as array list starts with index 0) of array list.

3) [**remove**](https://beginnersbook.com/2013/12/java-arraylist-removeobject-method-example/)**(Object o)**: Removes the object o from the ArrayList.

obj.remove("Chaitanya");

This statement will remove the string “Chaitanya” from the ArrayList.

4) [**remove**](https://beginnersbook.com/2013/12/java-arraylist-remove-method-example/)**(int index)**: Removes element from a given index.

obj.remove(3);

It would remove the element of index 3 (4th element of the list – List starts with o).

5) [**set**](https://beginnersbook.com/2013/12/java-arraylist-set-method-example/)**(int index, Object o)**: Used for updating an element. It replaces the element present at the specified index with the object o.

obj.set(2, "Tom");

It would replace the 3rd element (index =2 is 3rd element) with the value Tom.

6)**int indexOf(Object o)**: Gives the index of the object o. If the element is not found in the list then this method returns the value -1.

int pos = obj.indexOf("Tom");

This would give the index (position) of the string Tom in the list.

7) **Object**[**get**](https://beginnersbook.com/2013/12/java-arraylist-get-method-example/)**(int index)**: It returns the object of list which is present at the specified index.

String str= obj.get(2);

This would return the string stored at 3rd position (index 2) and would be assigned to the string “str”. We are using string variable to store the get() result because the list is of string type. If the list is of int type then we can use int variable to store the returned element.

8) [**int size()**](https://beginnersbook.com/2013/12/how-to-find-length-of-arraylist-in-java/): It returns the size of the ArrayList (Number of elements of the list).

int numberofitems = obj.size();

9) **boolean**[**contains**](https://beginnersbook.com/2013/12/java-arraylist-contains-method-example/)**(Object o)**: It checks whether the given object o is present in the array list. If the element is found it returns true else it returns false.

obj.contains("Steve");

It would return true if the string “Steve” is present in the list else we would get false.

10) [**clear**](https://beginnersbook.com/2013/12/how-to-empty-an-arraylist-in-java/)**():** It is used for removing all the elements of the array list in one go. The below code will remove all the elements of ArrayList whose object is obj.

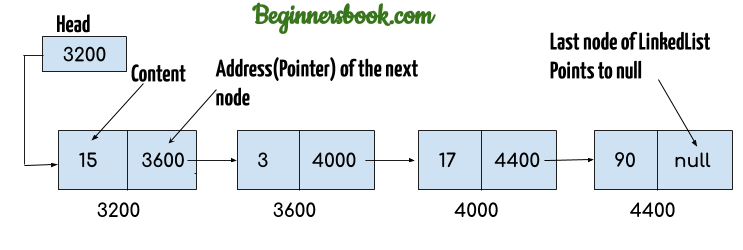
obj.clear();

# LinkedList in Java with Example

Similar to arrays in Java, **LinkedList is a linear data structure**. However LinkedList elements are not stored in contiguous locations like arrays, they are linked with each other using pointers. Each element of the LinkedList has the reference(address/pointer) to the next element of the LinkedList.

## LinkedList representation

Each element in the LinkedList is called the **Node**. Each Node of the LinkedList contains two items: 1) Content of the element 2) Pointer/Address/Reference to the Next Node in the LinkedList.

**This is how a LinkedList looks:**  


**Note:**  
1. **Head** of the LinkedList only contains the Address of the **First element** of the List.  
2. The Last element of the LinkedList contains **null** in the pointer part of the node because it is the end of the List so it doesn’t point to anything as shown in the above diagram.  
3. The diagram which is shown above represents a **singly linked list**. There is another complex type variation of LinkedList which is called **doubly linked list**, node of a doubly linked list contains three parts: 1) Pointer to the previous node of the linked list 2) content of the element 3) pointer to the next node of the linked list.

## Why do we need a Linked List?

You must be aware of the arrays which is also a linear data structure but **arrays have certain limitations such as:**

3) **Inserting an element in an array is performance wise expensive** as we have to shift several elements to make a space for the new element. For example:  
Let’s say we have an array that has following elements: 10, 12, 15, 20, 4, 5, 100, now if we want to insert a new element 99 after the element that has value 12 then we have to shift all the elements after 12 to their right to make space for new element.

Similarly **deleting an element** from the array is also a performance wise expensive operation because all the elements after the deleted element have to be shifted left.

**These limitations are handled in the Linked List by providing following features:**

3. Insert and delete operations in the Linked list are not performance wise expensive because adding and deleting an element from the linked list does’t require element shifting, only the pointer of the previous and the next node requires change.

## Hierarchy of LinkedList class in Java

## Java Linked List example of adding elements

In the following example we are using add(), addFirst() and addLast() methods to add the elements at the desired locations in the LinkedList, there are several such useful methods in the LinkedList class which I have mentioned at the end of this article.

package com.beginnersbook;

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

LinkedList<String> list=new LinkedList<String>();

//Adding elements to the Linked list

list.add("Steve");

list.add("Carl");

list.add("Raj");

//Adding an element to the first position

list.addFirst("Negan");

//Adding an element to the last position

list.addLast("Rick");

//Adding an element to the 3rd position

list.add(2, "Glenn");

//Iterating LinkedList

Iterator<String> iterator=list.iterator();

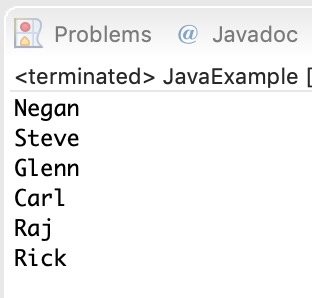
while(iterator.hasNext()){

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

}

}

}

**Output:**  


## Java example of removing elements from the LinkedList

In the following example we are checking out the few popular **remove methods** in the LinkedList that are used to remove elements from certain positions in the LinkedList

package com.beginnersbook;

import java.util.\*;

public class JavaExample{

public static void main(String args[]){

LinkedList<String> list=new LinkedList<String>();

//Adding elements to the Linked list

list.add("Steve");

list.add("Carl");

list.add("Raj");

list.add("Negan");

list.add("Rick");

//Removing First element

//Same as list.remove(0);

list.removeFirst();

//Removing Last element

list.removeLast();

//Iterating LinkedList

Iterator<String> iterator=list.iterator();

while(iterator.hasNext()){

System.out.print(iterator.next()+" ");

}

//removing 2nd element, index starts with 0

list.remove(1);

System.out.print("\nAfter removing second element: ");

//Iterating LinkedList again

Iterator<String> iterator2=list.iterator();

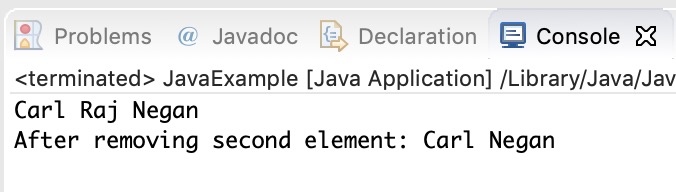
while(iterator2.hasNext()){

System.out.print(iterator2.next()+" ");

}

}

}

**Output:**  


## Example of LinkedList in Java

import java.util.\*;

public class LinkedListExample {

public static void main(String args[]) {

/\* Linked List Declaration \*/

LinkedList<String> linkedlist = new LinkedList<String>();

/\*add(String Element) is used for adding

\* the elements to the linked list\*/

linkedlist.add("Item1");

linkedlist.add("Item5");

linkedlist.add("Item3");

linkedlist.add("Item6");

linkedlist.add("Item2");

/\*Display Linked List Content\*/

System.out.println("Linked List Content: " +linkedlist);

/\*Add First and Last Element\*/

linkedlist.addFirst("First Item");

linkedlist.addLast("Last Item");

System.out.println("LinkedList Content after addition: " +linkedlist);

/\*This is how to get and set Values\*/

Object firstvar = linkedlist.get(0);

System.out.println("First element: " +firstvar);

linkedlist.set(0, "Changed first item");

Object firstvar2 = linkedlist.get(0);

System.out.println("First element after update by set method: " +firstvar2);

/\*Remove first and last element\*/

linkedlist.removeFirst();

linkedlist.removeLast();

System.out.println("LinkedList after deletion of first and last element: " +linkedlist);

/\* Add to a Position and remove from a position\*/

linkedlist.add(0, "Newly added item");

linkedlist.remove(2);

System.out.println("Final Content: " +linkedlist);

}

}

Output:

Linked List Content: [Item1, Item5, Item3, Item6, Item2]

LinkedList Content after addition: [First Item, Item1, Item5, Item3, Item6, Item2, Last Item]

First element: First Item

First element after update by set method: Changed first item

LinkedList after deletion of first and last element: [Item1, Item5, Item3, Item6, Item2]

Final Content: [Newly added item, Item1, Item3, Item6, Item2]

## Methods of LinkedList class:

Here I have mentioned the brief description of the LinkedList methods, I have covered each one of these methods in separate tutorials, links are provided at the end of this article.

For all the examples in the below methods, consider llistobj as a reference for LinkedList<String>.

LinkedList<String> llistobj  = new LinkedList<String>();

1) **boolean add(Object item)**: It adds the item at the end of the list.

llistobj.add("Hello");

It would add the string “Hello” at the end of the linked list.

2) **void add(int index, Object item)**: It adds an item at the given index of the the list.

llistobj.add(2, "bye");

This will add the string “bye” at the 3rd position( 2 index is 3rd position as index starts with 0).

3) **boolean addAll(Collection c)**: It adds all the elements of the specified collection c to the list. It throws NullPointerException if the specified collection is null. Consider the below example –

LinkedList<String> llistobj = new LinkedList<String>();

ArrayList<String> arraylist= new ArrayList<String>();

arraylist.add("String1");

arraylist.add("String2");

llistobj.addAll(arraylist);

This piece of code would add all the elements of ArrayList to the LinkedList.

4) **boolean addAll(int index, Collection c)**: It adds all the elements of collection c to the list starting from a give index in the list. It throws NullPointerException if the collection c is null and IndexOutOfBoundsException when the specified index is out of the range.

llistobj.add(5, arraylist);

It would add all the elements of the ArrayList to the LinkedList starting from position 6 (index 5).

5) **void addFirst(Object item)**: It adds the item (or element) at the first position in the list.

llistobj.addFirst("text");

It would add the string “text” at the beginning of the list.

6) **void addLast(Object item)**: It inserts the specified item at the end of the list.

llistobj.addLast("Chaitanya");

This statement will add a string “Chaitanya” at the end position of the linked list.

7) **void clear()**: It removes all the elements of a list.

llistobj.clear();

8) **Object clone()**: It returns the copy of the list.

For e.g. My linkedList has four items: text1, text2, text3 and text4.

Object str= llistobj.clone();

System.out.println(str);

Output: The output of above code would be:

[text1, text2, text3, text4]

9) **boolean contains(Object item)**: It checks whether the given item is present in the list or not. If the item is present then it returns true else false.

boolean var = llistobj.contains("TestString");

It will check whether the string “TestString” exist in the list or not.

10) **Object get(int index)**: It returns the item of the specified index from the list.

Object var = llistobj.get(2);

It will fetch the 3rd item from the list.

11) **Object getFirst()**: It fetches the first item from the list.

Object var = llistobj.getFirst();

12) **Object getLast()**: It fetches the last item from the list.

Object var= llistobj.getLast();

13) **int indexOf(Object item)**: It returns the index of the specified item.

llistobj.indexOf("bye");

14) **int lastIndexOf(Object item)**: It returns the index of last occurrence of the specified element.

int pos = llistobj.lastIndexOf("hello);

integer variable pos will be having the index of last occurrence of string “hello”.

15) **Object poll()**: It returns and removes the first item of the list.

Object o = llistobj.poll();

16) **Object pollFirst()**: same as poll() method. Removes the first item of the list.

Object o = llistobj.pollFirst();

17) **Object pollLast()**: It returns and removes the last element of the list.

Object o = llistobj.pollLast();

18) **Object remove()**: It removes the first element of the list.

llistobj.remove();

19) **Object remove(int index)**: It removes the item from the list which is present at the specified index.

llistobj.remove(4);

It will remove the 5th element from the list.

20) **Object remove(Object obj)**: It removes the specified object from the list.

llistobj.remove("Test Item");

21) **Object removeFirst()**: It removes the first item from the list.

llistobj.removeFirst();

22) **Object removeLast()**: It removes the last item of the list.

llistobj.removeLast();

23) **Object removeFirstOccurrence(Object item)**: It removes the first occurrence of the specified item.

llistobj.removeFirstOccurrence("text");

It will remove the first occurrence of the string “text” from the list.

24) **Object removeLastOccurrence(Object item)**: It removes the last occurrence of the given element.

llistobj.removeLastOccurrence("String1);

It will remove the last occurrence of string “String1”.

25) **Object set(int index, Object item)**: It updates the item of specified index with the give value.

llistobj.set(2, "Test");

It will update the 3rd element with the string “Test”.

26)**int size()**: It returns the number of elements of the list.

llistobj.size();

# How to loop LinkedList in Java

In the last tutorial we discussed LinkedList and it’s methods with example. Here we will see how to loop/iterate a LinkedList. There are four ways in which a LinkedList can be iterated –

1. For loop
2. Advanced For loop
3. Iterator
4. While Loop

#### Example:

In this example we have a LinkedList of String Type and we are looping through it using all the four mentioned methods.

package beginnersbook.com;

import java.util.\*;

public class LinkedListExample {

public static void main(String args[]) {

/\*LinkedList declaration\*/

LinkedList<String> linkedlist=new LinkedList<String>();

linkedlist.add("Apple");

linkedlist.add("Orange");

linkedlist.add("Mango");

/\*for loop\*/

System.out.println("\*\*For loop\*\*");

for(int num=0; num<linkedlist.size(); num++)

{

System.out.println(linkedlist.get(num));

}

/\*Advanced for loop\*/

System.out.println("\*\*Advanced For loop\*\*");

for(String str: linkedlist)

{

System.out.println(str);

}

/\*Using Iterator\*/

System.out.println("\*\*Iterator\*\*");

Iterator i = linkedlist.iterator();

while (i.hasNext()) {

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

}

/\* Using While Loop\*/

System.out.println("\*\*While Loop\*\*");

int num = 0;

while (linkedlist.size() > num) {

System.out.println(linkedlist.get(num));

num++;

}

}

}

Output:

\*\*For loop\*\*

Apple

Orange

Mango

\*\*Advanced For loop\*\*

Apple

Orange

Mango

\*\*Iterator\*\*

Apple

Orange

Mango

\*\*While Loop\*\*

Apple

Orange

Mango

# Vector in Java

Vector implements List Interface. Like ArrayList it also maintains insertion order but it is rarely used in non-thread environment as it is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.

#### Three ways to create vector class object:

**Method 1:**

Vector vec = new Vector();

It creates an empty Vector with the default initial capacity of 10. It means the Vector will be re-sized when the 11th elements needs to be inserted into the Vector. Note: By default vector doubles its size. i.e. In this case the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20 (double of default capacity 10).

**Method 2:**  
Syntax: Vector object= new Vector(int initialCapacity)

Vector vec = new Vector(3);

It will create a Vector of initial capacity of 3.

**Method 3:**  
Syntax:

Vector object= new vector(int initialcapacity, capacityIncrement)

Example:

Vector vec= new Vector(4, 6)

Here we have provided two arguments. The initial capacity is 4 and capacityIncrement is 6. It means upon insertion of 5th element the size would be 10 (4+6) and on 11th insertion it would be 16(10+6).

## Complete Example of Vector in Java:

import java.util.\*;

public class VectorExample {

public static void main(String args[]) {

/\* Vector of initial capacity(size) of 2 \*/

Vector<String> vec = new Vector<String>(2);

/\* Adding elements to a vector\*/

vec.addElement("Apple");

vec.addElement("Orange");

vec.addElement("Mango");

vec.addElement("Fig");

/\* check size and capacityIncrement\*/

System.out.println("Size is: "+vec.size());

System.out.println("Default capacity increment is: "+vec.capacity());

vec.addElement("fruit1");

vec.addElement("fruit2");

vec.addElement("fruit3");

/\*size and capacityIncrement after two insertions\*/

System.out.println("Size after addition: "+vec.size());

System.out.println("Capacity after increment is: "+vec.capacity());

/\*Display Vector elements\*/

Enumeration en = vec.elements();

System.out.println("\nElements are:");

while(en.hasMoreElements())

System.out.print(en.nextElement() + " ");

}

}

Output:

Size is: 4

Default capacity increment is: 4

Size after addition: 7

Capacity after increment is: 8

Elements are:

Apple Orange Mango Fig fruit1 fruit2 fruit3

# HashSet in Java With Examples

This class implements the Set interface, backed by a hash table (actually a HashMap instance).

**Points to Note about HashSet:**

1. HashSet **internally uses Hashtable** data structure.
2. HashSet **doesn’t maintain any order**, the elements would be returned in any random order.
3. HashSet **doesn’t allow duplicates**. If you try to add a duplicate element in HashSet, the old value would be overwritten.
4. HashSet **allows null values**, however if you insert more than one nulls, it would override the previous null value.
5. HashSet is **non-synchronized**. However it can be synchronized explicitly like this: Set s = Collections.synchronizedSet(new HashSet(...));

## A Simple Example of HashSet in Java

Let’s see a simple HashSet example, where we are adding few string elements to HashSet and then iterating the HashSet to print the elements.

import java.util.HashSet;

public class JavaExample {

public static void main(String args[]) {

// HashSet declaration

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

// Adding elements to the HashSet

hSet.add("Cricket");

hSet.add("Hockey");

hSet.add("Basketball");

System.out.println("HashSet Elements: ");

// Iterating HashSet

for(String s: hSet){

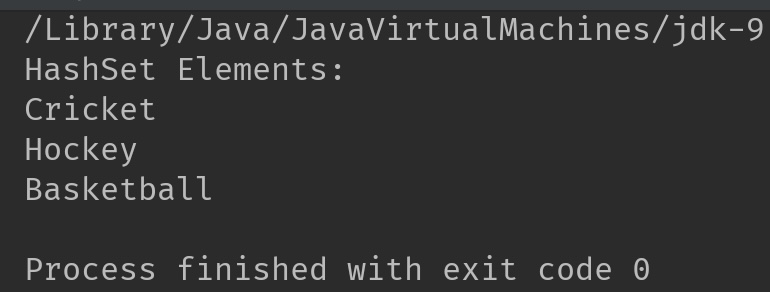
System.out.println(s);

}

}

}

**Output:**



## HashSet class Hierarchy

HashSet class extends AbstractSet class. The AbstractSet class implements Set interface, which extends Collection interface. This hierarchy can be represented as follows:

HashSet -> AbstractSet -> Set -> Collection -> Iterable

## HashSet Declaration

HashSet class belongs to java.util package. The declaration of HashSet in Java is:

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

## Initial Capacity and Load Factor:

**Initial capacity** represents the initial data buckets allocated to HashSet, this automatically increases when HashSet gets full.

**Load factor** measures the load of HashSet, it represents how much the HashSet is full. A load factor of .60 means that when HashSet is 60% full, the capacity of HashSet is automatically increased.

Number of element in HashSet

Load Factor Of HashSet = ----------------------------

Size of the HashSet

## Constructors of Java HashSet Class

|  |  |
| --- | --- |
| Constructor | Description |
| HashSet() | It builds an empty HashSet with initial capacity of 16 and load factor of .75. Example: HashSet<String> hSet = HashSet<>(); |
| HashSet(int initialCapacity) | It builds an empty HashSet with the specified initial capacity. The default load factor remains .75. |
| HashSet(int initialCapacity, float loadFactor) | It builds an empty HashSet with the specified initial capacity and load factor. |
| HashSet(Collection) | This doesn’t create an empty HashSet. It creates the HashSet with the elements copied from the passed Collection. |

## Java HashSet Examples

Let’s see few examples of HashSet in Java.

### 1. Adding duplicate elements

HashSet overrides duplicate values.

import java.util.HashSet;

public class HashSetExample {

public static void main(String args[]) {

// HashSet declaration

HashSet<String> hset =

new HashSet<String>();

// Adding elements to the HashSet

hset.add("Apple");

hset.add("Mango");

hset.add("Grapes");

hset.add("Orange");

hset.add("Fig");

//Addition of duplicate elements

hset.add("Apple");

hset.add("Mango");

//Addition of null values

hset.add(null);

hset.add(null);

//Displaying HashSet elements

System.out.println(hset);

}

}

**Output:**

[null, Mango, Grapes, Apple, Orange, Fig]

As you can see there all the duplicate values are not present in the output including the duplicate null value.

### 2. Removing elements

import java.util.HashSet;

public class JavaExample {

public static void main(String args[]) {

// HashSet declaration

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

// Adding elements to the HashSet

hSet.add("AA");

hSet.add("BB");

hSet.add("CC");

hSet.add("DD");

hSet.add("EE");

//removing elements

hSet.remove("EE");

hSet.remove("CC");

System.out.println("HashSet Elements: ");

// Iterating HashSet

for(String s: hSet){

System.out.println(s);

}

}

}

**Output:**

HashSet Elements:

AA

BB

DD

### 3. Adding elements from other Collection

Here, we are adding [ArrayList](https://beginnersbook.com/2013/12/java-arraylist/) elements to HashSet

import java.util.\*;

public class JavaExample {

public static void main(String args[]) {

//ArrayList declaration and and adding elements

ArrayList<String> arrList=new ArrayList<>();

arrList.add("AA");

arrList.add("BB");

arrList.add("CC");

//copying ArrayList elements to HashSet

HashSet<String> hSet=new HashSet(arrList);

//adding another element to HashSet after copy

hSet.add("DD");

System.out.println("HashSet elements: ");

Iterator<String> it= hSet.iterator();

while(it.hasNext())

{

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

}

}

}

**Output:**

HashSet elements:

AA

BB

CC

DD

## HashSet Methods:

1. **boolean add(Element  e)**: It adds the element e to the list.
2. **void clear()**: It removes all the elements from the list.
3. **Object clone()**: This method returns a shallow copy of the HashSet.
4. **boolean contains(Object o)**: It checks whether the specified Object o is present in the list or not. If the object has been found it returns true else false.
5. **boolean isEmpty()**: Returns true if there is no element present in the Set.
6. **int size()**: It gives the number of elements of a Set.
7. **boolean(Object o)**: It removes the specified Object o from the Set.

# How to Iterate over a Set/HashSet

BY CHAITANYA SINGH

There are following two ways to iterate through HashSet:  
1) Using Iterator  
2) Without using Iterator

#### Example 1: Using Iterator

import java.util.HashSet;

import java.util.Iterator;

class IterateHashSet{

public static void main(String[] args) {

// Create a HashSet

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

//add elements to HashSet

hset.add("Chaitanya");

hset.add("Rahul");

hset.add("Tim");

hset.add("Rick");

hset.add("Harry");

Iterator<String> it = hset.iterator();

while(it.hasNext()){

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

}

}

}

**Output:**

Chaitanya

Rick

Harry

Rahul

Tim

### Example 2: Iterate without using Iterator

import java.util.HashSet;

import java.util.Set;

class IterateHashSet{

public static void main(String[] args) {

// Create a HashSet

Set<String> hset = new HashSet<String>();

//add elements to HashSet

hset.add("Chaitanya");

hset.add("Rahul");

hset.add("Tim");

hset.add("Rick");

hset.add("Harry");

for (String temp : hset) {

System.out.println(temp);

}

}

}

**Output:**

Chaitanya

Rick

Harry

Rahul

Tim

# How to convert a HashSet to a TreeSet

#### Description

Program to convert a HashSet to a TreeSet

#### Program

Here is the complete code for HashSet to TreeSet conversion. We have a HashSet of Strings and we are creating a TreeSet of strings by copying all the elements of HashSet to TreeSet.

import java.util.HashSet;

import java.util.TreeSet;

import java.util.Set;

class ConvertHashSettoTreeSet{

public static void main(String[] args) {

// Create a HashSet

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

//add elements to HashSet

hset.add("Element1");

hset.add("Element2");

hset.add("Element3");

hset.add("Element4");

// Displaying HashSet elements

System.out.println("HashSet contains: "+ hset);

// Creating a TreeSet of HashSet elements

TreeSet<String> tset = new TreeSet<String>(hset);

// Displaying TreeSet elements

System.out.println("TreeSet contains: ");

for(String temp : tset){

System.out.println(temp);

}

}

}

**Output:**

HashSet contains: [Element1, Element2, Element3, Element4]

TreeSet contains:

Element1

Element2

Element3

Element4

# Convert HashSet to a List/ArrayList

learning how to convert a HashSet to a List (ArrayList).

#### Program

Here we have a HashSet of String elements and we are creating an ArrayList of Strings by copying all the elements of HashSet to ArrayList. Following is the complete code:

import java.util.HashSet;

import java.util.List;

import java.util.ArrayList;

class ConvertHashSetToArrayList{

public static void main(String[] args) {

// Create a HashSet

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

//add elements to HashSet

hset.add("Steve");

hset.add("Matt");

hset.add("Govinda");

hset.add("John");

hset.add("Tommy");

// Displaying HashSet elements

System.out.println("HashSet contains: "+ hset);

// Creating a List of HashSet elements

List<String> list = new ArrayList<String>(hset);

// Displaying ArrayList elements

System.out.println("ArrayList contains: "+ list);

}

}

**Output:**

HashSet contains: [Tommy, Matt, Steve, Govinda, John]

ArrayList contains: [Tommy, Matt, Steve, Govinda, John]

# LinkedHashSet Class in Java with Example

Earlier we have shared tutorials on [HashSet](https://beginnersbook.com/2013/12/hashset-class-in-java-with-example/) and [TreeSet](https://beginnersbook.com/2013/12/treeset-class-in-java-with-example/). [LinkedHashSet](https://docs.oracle.com/javase/6/docs/api/java/util/LinkedHashSet.html) is also an implementation of Set interface, it is similar to the HashSet and TreeSet except the below mentioned differences:

1. HashSet doesn’t maintain any kind of order of its elements.
2. TreeSet sorts the elements in ascending order.
3. LinkedHashSet maintains the insertion order. Elements gets sorted in the same sequence in which they have been added to the Set.

## Example of LinkedHashSet:

import java.util.LinkedHashSet;

public class LinkedHashSetExample {

public static void main(String args[]) {

// LinkedHashSet of String Type

LinkedHashSet<String> lhset = new LinkedHashSet<String>();

// Adding elements to the LinkedHashSet

lhset.add("Z");

lhset.add("PQ");

lhset.add("N");

lhset.add("O");

lhset.add("KK");

lhset.add("FGH");

System.out.println(lhset);

// LinkedHashSet of Integer Type

LinkedHashSet<Integer> lhset2 = new LinkedHashSet<Integer>();

// Adding elements

lhset2.add(99);

lhset2.add(7);

lhset2.add(0);

lhset2.add(67);

lhset2.add(89);

lhset2.add(66);

System.out.println(lhset2);

}

}

**Output:**

[Z, PQ, N, O, KK, FGH]

[99, 7, 0, 67, 89, 66]

Observe the output: Both types of LinkedHashSet have preserved the insertion order.

# TreeSet Class in Java with example

TreeSet is similar to [HashSet](https://beginnersbook.com/2013/12/hashset-class-in-java-with-example/) except that it sorts the elements in the ascending order while HashSet doesn’t maintain any order. TreeSet allows null element but like HashSet it doesn’t allow. Like most of the other collection classes this class is also not synchronized, however it can be synchronized explicitly like this: SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));

## TreeSet Example:

In this example we have two TreeSet (TreeSet<String> & TreeSet<Integer>). We have added the values to both of them randomly however the result we got is sorted in ascending order.

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String args[]) {

// TreeSet of String Type

TreeSet<String> tset = new TreeSet<String>();

// Adding elements to TreeSet<String>

tset.add("ABC");

tset.add("String");

tset.add("Test");

tset.add("Pen");

tset.add("Ink");

tset.add("Jack");

//Displaying TreeSet

System.out.println(tset);

// TreeSet of Integer Type

TreeSet<Integer> tset2 = new TreeSet<Integer>();

// Adding elements to TreeSet<Integer>

tset2.add(88);

tset2.add(7);

tset2.add(101);

tset2.add(0);

tset2.add(3);

tset2.add(222);

System.out.println(tset2);

}

}

Output: You can see both the TreeSet have been sorted in ascending order implicitly.

[ABC, Ink, Jack, Pen, String, Test]

[0, 3, 7, 88, 101, 222]

# 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.

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

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

## 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 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.

### 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

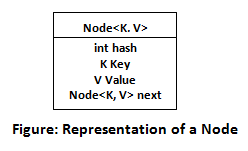
Working of HashMap in Java

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).

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

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

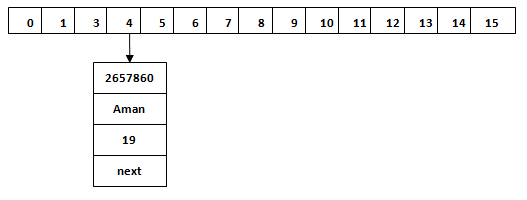
Index minimizes the size of the array. The Formula for calculating the index is:

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.

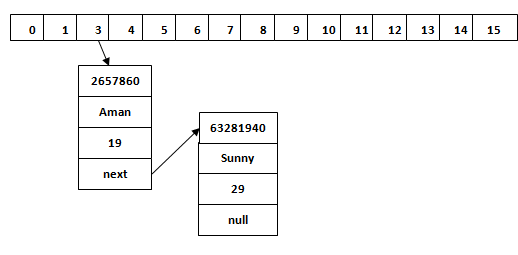


Hash Collision

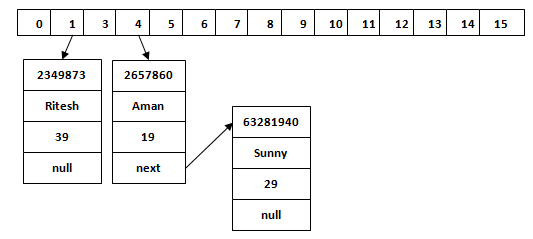
This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

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.

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:

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

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

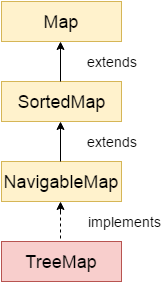
1 Details:

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.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

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

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 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.

## Java Collections Example

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

Output:

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

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

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

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

## Java Collections Example: max()

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

Output:

Value of maximum element from the collection: 67

## Java Collections Example: min()

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

Output:

Value of minimum element from the collection: 8

# Sorting in Collection

We can sort the elements of:

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

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

### Method of Collections class for sorting List elements

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

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

### Example to sort string objects

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

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

Mukesh

Saurav

Tahir

Viru

### Example to sort string objects in reverse order

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

Viru

Tahir

Saurav

Mukesh

### Example to sort Wrapper class objects

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

101

201

230

### Example to sort user-defined class objects

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

Mukesh

Saurav

Tahir

Viru

# 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.

Play Videox[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

### 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