## What is Collection in Java

A collection is a group of objects. In Java, these objects are called elements of the collection.

## Types of Objects Stored in Collection (Container) Object

**1. Homogeneous objects:**

Homo means same. Homogeneous objects are a group of multiple objects that belong to the same class.

For example, suppose we have created three objects Student s1, Student s2, and Student s3 of the same class ‘Student’. Since these three objects belong to the same class that’s why they are called homogeneous objects.

**2. Heterogeneous objects:**

Hetero means different. Heterogeneous objects are a group of different objects that belong to different classes.

For example, suppose we have created two different objects of different classes such as one object Student s1, and another one object Employee e1. Here, student and employee objects together are called a collection of heterogeneous objects.

These objects can also be further divided into two types. They are as follows:

**1. Duplicate objects:**

The multiple objects of a class that contains the same data are called duplicate objects. For example, suppose we create two person objects Person p1 and Person p2. Both of these objects have the same data.

Person p1 = new Person( "abc");

Person p2 = new Person("abc");

Since the above two objects have the same data “abc” therefore, these are called duplicate objects.

**2. Unique objects:**

The multiple objects of a class that contains different data are called unique objects. For example:

Person p1 = new Person("abcd");

Person p2 = new Person("abcde");

## What is Collections Framework in Java?

A framework in java is a set of several classes and interfaces which provide a ready-made architecture.

Collections framework in Java supports two types of containers:

* One for storing a collection of elements (objects), that is simply called a collection.
* The other, for storing key/value pairs, which is called a map.

## Difference between Arrays & Collections in Java

1. Arrays are fixed in size but collections are growable in nature. We can increase or decrease size.

2. Arrays are not recommended to use with respect to memory whereas collections are recommended to use with respect to memory.

3. Arrays are recommended to use with respect to performance but collections are not recommended to use with respect to performance.

4. Arrays can store only homogeneous data elements (similar type of data) but collections can hold both homogeneous and heterogeneous elements.

5. Arrays do not support any method but collections support various kinds of methods.

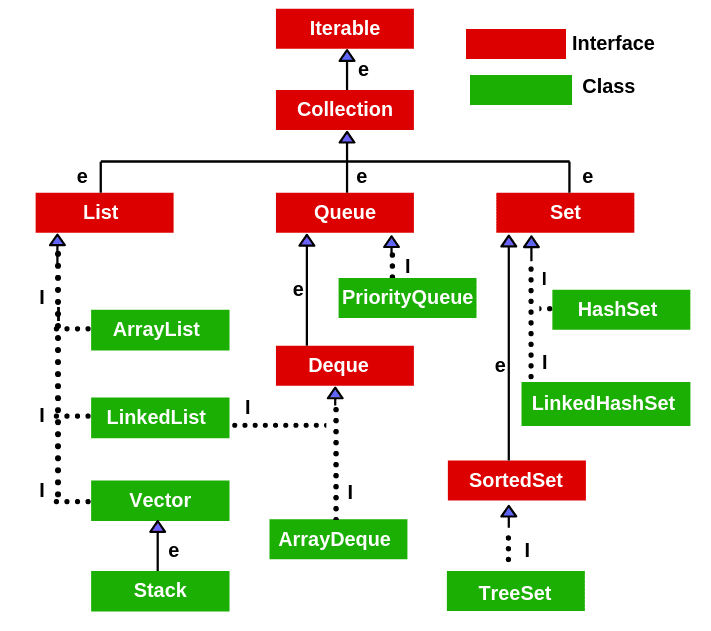
6. Arrays can have both hold primitives and object types but collections can hold only objects but not primitive.

## Advantage of Collections Framework in Java

1. The collections framework reduces the development time and the burden of designers, programmers, and users.
2. Your code is easier to maintain because it provides useful data structure and interfaces which reduce programming efforts.
3. The size of the container is growable in nature.
4. It implements high-performance of useful data structures and algorithms that increase the performance.
5. It enables software reuse.

**Collection Hierarchy in Java**

The hierarchy of the entire collection framework consists of four core interfaces such as Collection, List, Set, Map, and two specialized interfaces named SortedSet and SortedMap for sorting.



## Collection Interface in Java

**#** The basic interface of the collections framework is the Collection interface which is the root interface of all collections in the API (Application programming interface).

**#** The Collection interface extends the Iterable interface. The iterable interface has only one method called iterator(). The function of the iterator method is to return the iterator object. Using this iterator object, we can iterate over the elements of the collection.

**#** [List](https://www.scientecheasy.com/2020/09/java-list-interface.html/), Queue, and [Set](https://www.scientecheasy.com/2020/09/java-set.html/) have three component which extends the Collection interface. A map is not inherited by Collection interface.

**List Interface**

➲ This interface represents a collection of elements whose elements are arranged sequentially ordered.

➲ List maintains an order of elements means the order is retained in which we add elements, and the same sequence we will get while retrieving elements.

➲ We can insert elements into the list at any location. The list allows storing duplicate elements in Java.

➲ [ArrayList](https://www.scientecheasy.com/2020/09/arraylist-in-java.html/), [vector](https://www.scientecheasy.com/2020/09/vector-in-java.html/), and [LinkedList](https://www.scientecheasy.com/2020/09/java-linkedlist.html/) are three concrete subclasses that implement the list interface.

**Set Interface**

➲ This interface represents a collection of elements that contains unique elements. i.e, It is used to store the collection of unique elements.

➲ Set interface does not maintain any order while storing elements and while retrieving, we may not get the same order as we put elements.  All the elements in a set can be in any order.

➲ Set does not allow any duplicate elements.

➲ HashSet, LinkedHashSet, TreeSet classes implements the set interface and sorted interface extends a set interface.

➲ It can be iterated by using Iterator but cannot be iterated using ListIterator.

**SortedSet Interface**

➲ This interface extends a set whose iterator transverse its elements according to their natural ordering.

➲ TreeSet implements the sorted interface.

**Queue Interface**

➲ A queue is an ordered of the homogeneous group of elements in which new elements are added at one end(rear) and elements are removed from the other end(front). Just like a queue in a supermarket or any shop.

➲ This interface represents a special type of list whose elements are removed only from the head.

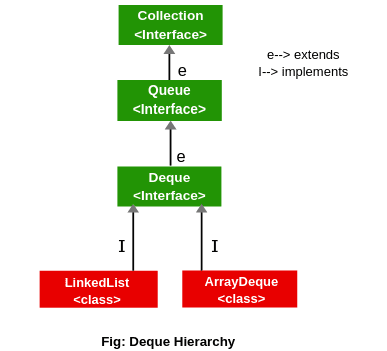
➲ LinkedList, Priority queue, ArrayQueue, Priority Blocking Queue, and Linked Blocking Queue are the concrete subclasses that implement the queue interface.

**Deque Interface**

➲ A deque (double-ended queue) is a sub-interface of queue interface. It is usually pronounced “deck”.

➲ This interface was added to the collection framework in Java SE 6.

➲ Deque interface extends the queue interface and uses its method to implement deque.

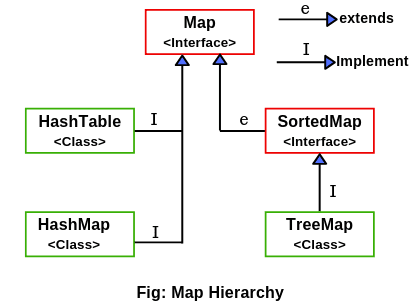


It is a linear collection of elements in which elements can be inserted and removed from either end. i.e, it supports insertion and removal at both ends of an object of a class that implements it.

➲ LinkedList and ArrayDeque classes implement the Deque interface.

**Map Interface**

➲ Map interface is not inherited by the collection interface. It represents an object that stores and retrieves elements in the form of a Key/Value pairs and their location within the Map are determined by a Key.



➲Map uses a hashing technique for storing key-value pairs.

➲ It doesn’t allow to store the duplicate keys but duplicate values are allowed.

➲ HashMap, HashTable, LinkedHashMap, TreeMap classes implements Map interface.

**SortedMap Interface**

➲ This interface represents a Map whose elements are stored in their natural ordering. It extends the Map interface which in turn is implemented by TreeMap classes.

## Methods of Collection Interface in Java

The Collection interface consists of a total of fifteen methods for manipulating elements in the collection. They are as follows:

1. **add():** This method is used to add or insert an element in the collection. The general syntax for add() method is as follow:

**add(Object element) : boolean**

If the element is added to the collection, it will return true otherwise false, if the element is already present and the collection doesn’t allow duplicates.

2. **addAll():** This method adds a collection of elements to the collection. It returns true if the elements are added otherwise returns false. The general syntax for this method is as follows:

**addAll(Collection c) : Boolean**

**3. clear():** This method clears or removes all the elements from the collection. The general form of this method is as follows:

**clear() : void**

This method returns nothing.

4. **equals():** It checks for equality with another object. The general form is as follows:

**equals(Object element) : Boolean**

5. isEmpty(): It returns true if a collection is empty. That is, this method returns true if the collection contains no elements.

**isEmpty() : boolean**

6. iterator(): It returns an iterator. The general form is given below:

**iterator() : Iterator**

7. remove(): It removes a specified element from the collection. The general syntax is given below:

**remove(Object element) : boolean**

This method returns true if the element was removed. Otherwise, it returns false.

8. removeAll(): The removeAll() method removes all elements from the collection. It returns true if all elements are removed otherwise returns false.

**removeAll(Collection c) : boolean**

9. retainAll(): This method is used to remove all elements from the collection except the specified collection. It returns true if all the elements are removed otherwise returns false.

**retainAll(Collection c) : boolean**

10. size(): The size() method returns the total number of elements in the collection. Its return type is an integer. The general syntax is given below:

**size() : int**

11. toArray(): It returns the elements of a collection in the form of an array. The array elements are copies of the collection elements.

**toArray() : Object[]**

# Java List Interface | Methods, Example

➲ A **list in Java** is a collection for storing elements in sequential order. Sequential order means the first element, followed by the second element, followed by the third element, and so on.

➲ Java list is a sub-interface of the collection interface that is available in java.util package. Sub interface means an interface that extends another interface is called sub interface. Here, the list interface extends the collection interface.

➲ It is an ordered collection where elements are maintained by index positions because it uses an index-based structure. In the list interface, the order is retained in which we add elements. We will also get the same sequence while retrieving elements.

➲ It is used to store a collection of elements where duplicate elements are allowed.

➲ List interface in java has four concrete subclasses. They are ArrayList, LinkedList, Vector, and Stack. These four subclasses implements the list interface.

ArrayList and LinkedList are widely used in Java programs to create a list. The Vector class is deprecated since JDK 5.

## Features of List Interface in Java

1. The list allows storing duplicate elements in Java. JVM differentiates duplicate elements by using ‘index’. Index refers to the position of a certain object in an array. It always starts at zero.
2. In the list, we can add an element at any position.
3. 3. It maintains insertion order. i.e. List can preserve the insertion order by using the index.
4. 4. It allows for storing many null elements.
5. 5. Java list uses a resizable array for its implementation. Resizable means we can increase or decrease the size of the array.
6. 6. Except for LinkedList, ArrayList, and Vector is an indexed-based structure.
7. 7. It provides a special Iterator called a ListIterator that allows accessing the elements in the forward direction using hasNext() and next() methods.
8. In the reverse direction, it accesses elements using hasPrevious() and previous() methods. We can add, remove elements of the collection, and can also replace the existing elements with the new element using ListIterator.

## How to create a List

To create a list in java, we can use one of its two concrete subclasses: ArrayList, and LinkedList. We will use ArrayList to create a list and test methods provided by list interface in the program section.

List p = new ArrayList();

List q = new LinkedList();

List r = new Vector();

List s = new Stack();

## How to create Generic List Object in Java

## 1. List<data type> list = new ArrayList<data type>(); // General sysntax.

## For example:

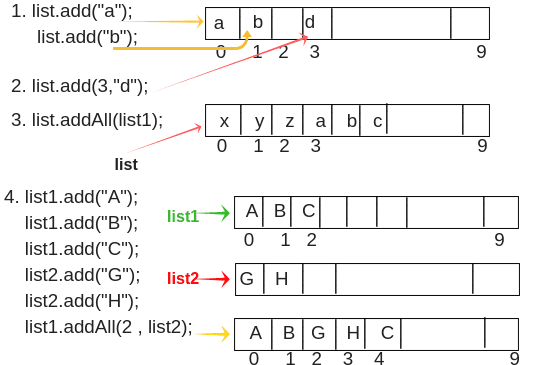
## a. List<String> list = new ArrayList<String>(); // Creating a list of objects of String type using ArrayList.

## b. List<Integer> list = new LinkedList<Integer>(); Creating a list of objects of Integer type using LinkedList.

## c. List<String> list1 = new LinkedList<String>();

## Java List Methods

1. **boolean add(Object o):** It starts to add the specified element from zero location. If the element is already present at zero location, it will add the next element in one position.



The return type of add() method is boolean and input type is an object.  
**2. void add(int index, Object o):** This method adds/inserts the specified element at a particular position in the list. For example, suppose we want to add element “d” at 3rd position, we will call add(int index, Object o ) method like this:

**list.add(3,"d");** // It will add element "d" at 3rd position as shown in figure.

**3. boolean addAll(Collection c):** Here, Collection c represents a group of elements. This method is used to add/insert a group of elements at the end of the last element.

For example, suppose we want to add three elements x, y, and z at positions 0, 1, and 2 respectively in a list.

list.add("x");

list.add("y");

list.add("z");

Now we will create a list of another group of elements like this:

list1.add("a");

list1.add("b");

list1.add("c");

list.addAll(list1); // It will add group of elements at the end of the last element in the list. The last element is z. So, after z, it will add list1 as shown in above figure.

4. **boolean addAll(int index, Collection c):** This method is used to add/insert a group of elements at a particular position in the list and shift the subsequent elements to the right by increasing their indices.

For example, suppose we want to add three elements at positions 1, 2, and 3 respectively in a list using list1 reference variable.

list1.add("A");

list1.add("B");

list1.add("C");

Now we will create a list of another group of elements using list2 reference variable.

list2.add("G");

list2.add("H");

Assume that we want to add this group of elements at position 2 of the list1. So, we will call addAll(int index, Collection c) method like this:

list1.addAll(2, list2); // You will see that element C is shifted to right at position 4 as shown in the figure.

**5. object remove(int index):** It is used to remove an element at a specified position in the list. For example, consider the above figure.

list1.remove(2);

**6. object get(int index):** This method is used to return element/object stored at a specified position in the list. The return type of get() method is Object and input type is int[index of List].

For example:

list1.get(2);

**7. int indexOf(Object o):** It is used to return the index of a particular element of the first occurrence in the list. If the element is not present in the list then it will return -1. It takes as an argument as an element and returns as an integer value of that element as it is index value.

list.indexOf("A"); // It will return integer value of an element "A" of first occurrence. i.e from zero position, not from position 9. So the output is 0.

**When to use List?**

There are the following points for using list in java application that should be to keep in mind. They are:

1. List can be used when we want to allow or store duplicate elements.  
2. It can be used when we want to store null elements.  
3. When we want to preserve my insertion order, we should go for list.

## List Example Programs

Program source code 1:

import java.util.ArrayList;

import java.util.List;

public class AddEx

{

public static void main(String[] args)

{

// Create a List.

List al = new ArrayList(); // Here, there is no use of generic. So, no type safety. We can add both integer and string elements.

// Adding elements using add() method with reference variable al.

al.add(10);

al.add(20);

al.add(30);

al.add(40);

al.add("Shubh");

// Adding element to 4th position.

al.add(4, 35);

// Adding element to 5th position.

al.add(5, 45);

System.out.println("Elements after adding: " +al);

}

}

Output:

Elements after adding: [10, 20, 30, 40, 35, 45, Shubh]

Program source code 2:

package listPrograms;

import java.util.ArrayList;

import java.util.List;

public class AddAllEx

{

public static void main(String[] args)

{

// Create a list1 of only String type. This means that Compiler will give errors if we try to put any elements other than String type.

List<String> al = new ArrayList<String>();

al.add("Apple");

al.add("Mango");

al.add("Orange");

al.add("Grapes");

System.out.println("List1 contain: " +al);

// Create another List2 of String type.

List<String> al2 = new ArrayList<String>();

al2.add("11");

al2.add("12");

al2.add("13");

System.out.println("List2 contain :-"+al2);

// Adding List2 in List1 at 2nd position(i.e index=2) using addAll() method.

al.addAll(2, al2);

System.out.println("List1 after adding List2 at 2nd position :-"+al);

}

}

package listPrograms;

import java.util.ArrayList;

import java.util.List;

public class IndexOfEx

{

public static void main(String[] args)

{

List al = new ArrayList();

al.add("AA");

al.add("BB");

al.add("CC");

al.add("DD");

al.add("EE");

al.add("FF");

// To find the Index of any particular element, use obj.indexOf(object o) method.

System.out.println("Index of CC: "+al.indexOf("CC"));

System.out.println("Index of FF: "+al.indexOf("FF"));

}

}

package listPrograms;

import java.util.ArrayList;

import java.util.List;

public class GetMethodEx

{

public static void main(String[] args)

{

List al = new ArrayList();

// Adding Element using reference variable al.

al.add("pen");

al.add("pencil");

al.add("ink");

al.add("notebook");

al.add("book");

al.add("paper");

// Now call get(int index) method to get elements from specified index and print them.

System.out.println("First Element: " +al.get(0));

System.out.println("Fourth Element: " +al.get(3));

}

# ArrayList in Java | ArrayList Methods, Example

**ArrayList in Java** is a resizable array that can grow or shrink in the memory whenever needed. It is dynamically created with an initial capacity.

Java ArrayList uses a dynamic array internally for storing the group of elements or data.

**Serializable Interface**

1. A serializable interface is a marker interface that is used to send the group of objects over the network. It is present in the java.io package.

2. It helps in sending the data from one class to another class. Usually, we use collections to hold and transfer objects from one place to another place.

To provide support for this requirement, every collections class already implements Serializable and Cloneable.

**Cloneable Interface**

1. A cloneable interface is present in java.lang package.

2. It is used to create exactly duplicate objects. When the data or group of objects came from the network, the receiver will create duplicate objects.

The process of creating exactly duplicate objects is known as cloning. It is a very common requirement for collection classes.

**Features of ArrayList in Java**

**1. Resizable-array:** ArrayList is a resizable array or growable array that means the size of ArrayList can increase or decrease in size at runtime. Once ArrayList is created, we can add any number of elements.

**2. Index-based structure:** It uses an index-based structure in java.

**3. Duplicate elements:** Duplicate elements are allowed in the array list.

**4. Null elements:** Any number of null elements can be added to ArrayList.

**5. Insertion order:** It maintains the insertion order in Java. That is insertion order is preserved.

**6. Heterogeneous objects:** Heterogeneous objects are allowed everywhere except TreeSet and TreeMap. Heterogeneous means different elements.

**7. Synchronized:** ArrayList is not synchronized. That means [multiple threads](https://www.scientecheasy.com/2020/08/creating-multiple-threads-in-java.html/) can use the same ArrayList objects simultaneously.

**8. Random Access:** ArrayList implements random access because it uses an index-based structure. Therefore, we can get, set, insert, and remove elements of the array list from any arbitrary position.

**9. Performance:** In ArrayList, manipulation is slow because if any element is removed from ArrayList, a lot of shifting takes place.

**ArrayList Methods in Java**

1. **boolean add(Object o):** This method is used to add an element at the end of array list. For example,  if you want to add an element at the end of the list, you simply call the add() method like this:

list.add("Shubh");

**boolean addAll(Collection c):** This method is used to add a group of elements in a particular collection at the end of the list. For example, suppose we have a group of elements in the list2 and want to add at the end of the list1, we will call this method like this:

list1.addAll(list2);

**boolean addAll(int index, Collection c):** This method is used to add a group of elements at a specified position in a list. For example:

list1.addAll(2, list2):

**void add(int index, Object o):** It is used to add an element at a particular position index in the list. For example:

list.add(3, "a");

**void addAll(int index, Object o):** It is used to add a specific element at a particular position in the list. For example, suppose we want to a specific element “Shubh” at a position 2 in the list, we will call add(int index, Object o) method like this:

list.add(2,"Shubh");

Program source code 1:

package ArrayListTest;

import java.util.ArrayList;

public class AddExample

{

public static void main(String[] args)

{

// Create an object of the non-generic ArrayList.

ArrayList al = new ArrayList(); // list 1 with default capacity 10.

al.add("A");

al.add("B");

al.add(20);

al.add("A");

al.add(null);

System.out.println(al);

// Create an object of another non-generic ArrayList.

ArrayList al1 = new ArrayList(); // List 2.

al1.add("a");

al1.add("b");

al1.add("c");

// Call addAll(Collection c) method using reference variable al to add all elements at the end of the list1.

al.addAll(al1);

System.out.println(al);

// Call addAll(int index, Collection c) method using reference variable al1 to add all elements at specified position 2.

al1.addAll(2, al);

System.out.println(al1);

}

}

**5. boolean remove(Object o):** It removes the first occurrence of the specified element from this list if it is present.

**6. void remove(int index):** This method removes the element from a particular position in the list. Look at the examples below.

list.remove("A");

list.remove(2); // It will remove element from position 2.

**7. void clear():** The clear() method is used to remove all elements from an array list.

**8. void set(int index, Object o):** The set() method replaces element at a particular position in the list with the specified element. For example, suppose we want to replace an element “A” at a position 2 with an element “a” in the list, we will have to call this method as:

list.set(2, "a");

Program source code 2:

package ArrayListTest;

import java.util.ArrayList;

public class RemoveEx

{

public static void main(String[] args)

{

// Create a generic Arraylist object of String type.

// This means the compiler will show an error if we try to put any other element than String.

ArrayList<String> al = new ArrayList<String>(); // Default capacity is 10.

// Adding elements of String type.

al.add("A");

al.add("B");

al.add("C");

al.add("D");

al.add(null);

al.add("D");

System.out.println(al);

// Call remove() method to remove element D.

al.remove("D"); // removes the first occurrence of the specified element D at position 3, not from the position 5.

System.out.println(al);

al.remove(3);

System.out.println(al);

// Call set method to replace the element D with a null element at position 3.

al.set(3, null);

System.out.println(al);

}

}

**9. Object get(int index):** It returns the element at the specified position in this list.

**10. int size():** It returns the number of elements of the list. Size means the number of elements present in the array list. Capacity means the capability to store elements.

**When to use ArrayList in Java?**

ArrayList can be used in an application program when

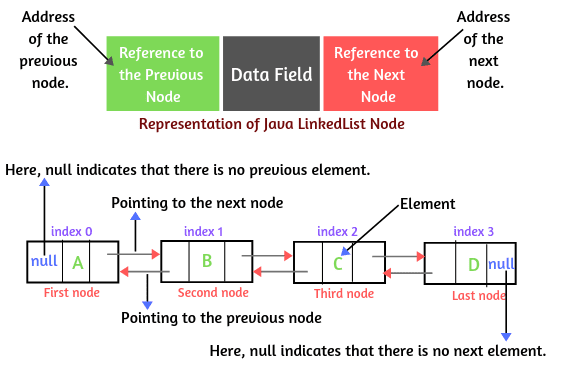
* We want to store duplicate elements.
* We want to store null elements.
* It is more preferred in Java when getting of the element is more as compared to adding and removing elements.
* We are not working in the multi-threading environment in Java because ArrayList is non-synchronized.

# LinkedList in Java | LinkedList Methods

➲ **LinkedList in Java** is a linear data structure that uses a doubly linked list internally to store a group of elements.

➲ Each node contains three fields: a data field that contains data stored in the node, left and right fields contain references or pointers that point to the previous and next nodes in the list.

A pointer indicates the addresses of the next node and the previous node. Elements in the linked list are called **nodes**.



**Features of LinkedList class**

1. The underlying data structure of LinkedList is a doubly LinkedList data structure. It is another concrete implementation of the List interface like an array list.

2. Java LinkedList class allows storing duplicate elements.

3. Null elements can be added to the linked list.

4. Heterogeneous elements are allowed in the linked list.

5. Java LinkedList is not synchronized. So, [multiple threads](https://www.scientecheasy.com/2020/08/creating-multiple-threads-in-java.html/) can access the same LinkedList object at the same time. Therefore, It is not thread-safe. Since LinkedList is not synchronized. Hence, its operation is faster.

6. Insertion and removal of elements in the LinkedList are fast because, in the linked list, there is no shifting of elements after each adding and removal. The only reference for next and previous elements changed.

7. LinkedList is the best choice if your frequent operation is insertion or deletion in the middle.

8. Retrieval (getting) of elements is very slow in LinkedList because it traverses from the beginning or ending to reach the element.

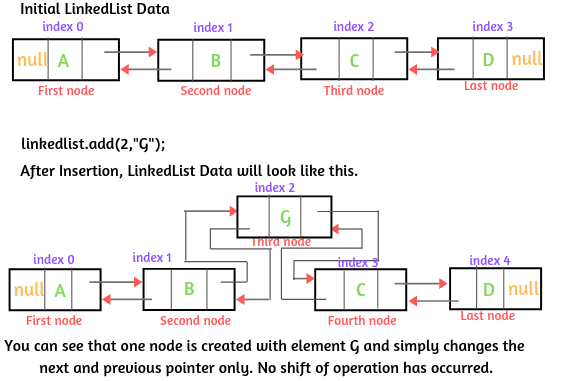
9. The LinkedList can be used as a “[stack](https://www.scientecheasy.com/2021/01/stack-in-java.html/)“. It has pop() and push() methods which make it function as a stack.

10. Java LinkedList does not implement random access interface. So, the element cannot be accessed (getting) randomly. To access the given element, we have to traverse from the beginning or ending to reach elements in the LinkedList.

11. We can iterate linked list elements by using [ListIterator](https://www.scientecheasy.com/2020/09/listiterator-in-java.html/).

**How does Insertion work in Java LinkedList?**

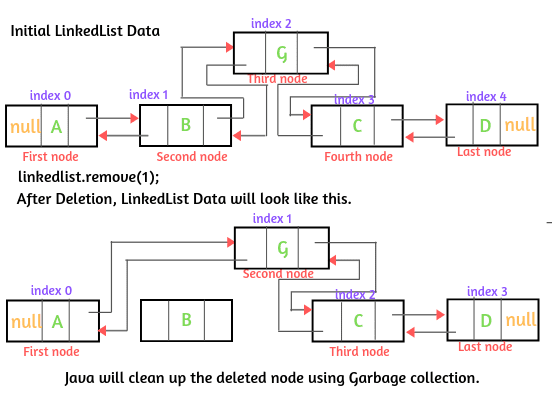
In the Java LinkedList, we can perform insertion (addition) operations without affecting any of data items already stored in the list. Let’s take an example to understand this concept.



1. Now we will perform insertion operation on this linked list. We will add an element G at index position 2 using add() method. The syntax for adding element G is as follow:

linkedlist.add(2,"G");

1. **How does Deletion work in Java LinkedList**

****

**linkedlist.remove(1);**

**LinkedList Methods in Java**

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1. | boolean add(Object o) | It is used to add the specified element to the end of a list. |
| 2. | void add(int index, Object o) | It is used to insert the specified element at the specified position in a list. |
| 3. | boolean addAll(Collection c) | It is used to add all of the elements in the specified collection to the end of this list |
| 4. | boolean addAll(int index, Collection c) | It is used to add all the elements of the specified collection at specified index position in the list. |
| 5. | void clear() | It is used to remove or delete all the elements from a list. |

# Set in Java | Set Methods, Example

A **set in Java** is an unordered collection of unique elements or objects. In other words, a set is a collection of elements that are not stored in a particular order.

We can add elements in a set and iterate over all elements in a set. It will grow dynamically when elements are stored in it. We can also check whether a particular element is present in the set

Java Set interface does not provide any get() method like List to retrieve elements. Therefore, the only way to take out elements from the set is to do using Iterator() method. But this method does not return elements from the set in any particular order.

Using the Iterator, we can traverse only in the forward direction from the first to last element. We cannot traverse over elements in the backward direction using iterator method.

**Java Set Implementation**

provides three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet.

**Set Methods in Java**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(Object o) | It is used to add the specified element in this set. |
| boolean addAll(Collection c) | This method adds all the elements in the given collection. |
| int size() | It is used to get the number of elements in the set. |
| boolean isEmpty() | This method checks that the set is empty or not. |

|  |  |
| --- | --- |
| boolean remove(Object o) | It is used to remove the specified element from this set. |
| boolean removeAll(Collection c) | It removes all the elements in the given collection from the set. |
| Iterator iterate() | It returns an Iterator over the elements in this set. |
| boolean equals(Object o) | It is used to compare the given element for equility in this set. |

**1. Adding Elements to Set:** The add() method returns true if set does not contain the specified element. If set already contains the specified element then it will return false.

Program source code 1:

package setProgram;

import java.util.HashSet;

import java.util.LinkedHashSet;

import java.util.Set;

public class SetExample1

{

public static void main(String[] args)

{

// Create a generic set object of type String.

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

int size = s.size();

System.out.println("Size before adding elements: " +size);

// Adding elements to set.

s.add("Orange"); // s.size()is 1.

s.add("Red"); // s.size() is 2.

s.add("Blue"); // s.size() is 3.

s.add("Yellow"); // s.size() is 4.

s.add("Green"); // Now s.size() is 5.

// Add duplicate elements in the set. These elements will be ignored by set due to not taking duplicate elements.

s.add("Red"); // s.size() is still 5 because we cannot add duplicate element.

s.add("Blue"); // s.size() is still 5 because we cannot add duplicate element.

System.out.println("Unordered Set Elements");

System.out.println(s);

// Check 'Black' element is present in the above set or not.

if(s.equals("Black"))

{

System.out.println("Black element is not present in set.");

}

else

{

s.add("Black");

System.out.println("Black is added successfully.");

System.out.println("Set Elements after adding black element");

System.out.println(s);

}

// Create another set object to add collection of elements to the above set.

Set<String> s2 = new LinkedHashSet<String>();

s2.add("White");

s2.add("Brown");

s2.add("Red"); // Duplicate element.

// Call addAll() method to add all the elements of the given collection.

s.addAll(s2);

System.out.println("Set Elements after adding elements from given collection");

System.out.println(s);

}

}

1. **Removing an Element from Set:** Let’s take a program where we will remove an element from set. We will also check the set is empty or not before adding elements in the list.

package setProgram;

import java.util.HashSet;

import java.util.Set;

public class SetExample2

{

public static void main(String[] args)

{

// Create a generic set object of type String.

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

// Check set is empty or not.

boolean check = s.isEmpty(); // Return type of this method is an boolean.

System.out.println("Is Set empty: " +check);

// Adding elements to set.

s.add("Orange");

s.add("Red");

s.add("Blue");

s.add("Yellow");

s.add("Green");

if(s.isEmpty())

{

System.out.println("Set is empty.");

}

else

{

System.out.println("Set is not empty.");

System.out.println("Elements in the Set");

System.out.println(s);

}

// Remove an element from set.

s.remove("Blue");

System.out.println("Set elements after removing");

System.out.println(s);

// Get the total number of set elements.

int size = s.size();

System.out.println("Total number of elements: " +size);

}

}

**When to use Set?**

1. If you want to represent a group of individual elements as a single entity where duplicates are not allowed and insertion order is not preserved then we should go for the Set.

2. If your requirement is to get unique elements, set is the best choice for this.

3. If you want to remove duplicate elements without maintaining the insertion order from the non-set collection, you should go for set.

import java.util.ArrayList;

import java.util.HashSet;

import java.util.List;

import java.util.Set;

public class SetExample4

{

public static void main(String[] args)

{

// Create a generic list object of type Integer.

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

int size = list.size();

System.out.println("Size before adding elements: " +size);

list.add(5);

list.add(10);

list.add(5);

list.add(15);

list.add(20);

list.add(10);

list.add(20);

list.add(30);

System.out.println("Original order of List Elements");

System.out.println(list);

Set<Integer> s = new HashSet<Integer>(list);

System.out.println("Unodered List Elements after removing duplicates.");

System.out.println(s);

}

}

# Java HashSet | Methods, Example

HashSet in Java is an unordered collection of elements (objects) that contains only unique elements. That is, it allows duplicate free elements to be stored.

**Features of HashSet**

There are several important features of Java HashSet that should be kept in mind. They are as follows:

1. The underlying data structure of HashSet is Hashtable. A hash table stores data by using a mechanism called **hashing**.

2. HashSet does not allow duplicate elements. If we try to add a duplicate element in HashSet, the older element would be overwritten.

3. It allows only one null element. If we try to add more than one null element, it would still return only one null value.

4. HashSet does not maintain any order in which elements are added. The order of the elements will be unpredictable. It will return in any random order.

5. It is much faster due to the use of hashing technique and gives constant-time performance for adding (insertion), retrieval, removal, contains, and size operations. Hashing provides constant execution time for methods like add(), remove(), contains(), and size() even for the large set.

6. HashSet class is not synchronized which means it is not thread-safe. Therefore, multiple threads can use the same HashSet object at the same time and will not give the deterministic final output.

**HashSet Class Methods in Java**

Java HashSet class does not define any additional methods. It inherits methods of its parent classes and methods of the implemented interface.

The various methods provided by its superclasses and interfaces are as follows:

**1. boolean add(Object o):** This method is used to add the specified element in this set. It returns true if set does not already contain a specified element.

**2. boolean addAll(Collection c):** This method is used to add a group of elements from another collection to the set.

**3. boolean remove(Object o):** This method is used to remove the specified element from the set.

To remove an element, set internally use equals() method for each element. If the element is found, element is removed from set and returns true.

If not found, false is returned. If the removal is not supported, you will get UnsupportedOperationException.

**4. void clear:** This method is used to remove all elements from a set. It returns nothing.

If the specified element is found in the set, it returns true otherwise false. The equality checking is done through the element’s equal method.

**6. int size():** If you wish to know how many elements are in a set, call size() method.

**7. boolean isEmpty():** If you wish to check whether HashSet contains elements or not. Call isEmpty() method.

**Java HashSet Example Program**

import java.util.HashSet;

public class HashSetExample1 {

public static void main(String[] args)

{

// Create a HashSet object.

HashSet<String> set = new HashSet<String>(); // An empty hash set.

// Adding elements to HashSet.

set.add("First");

set.add("Second");

set.add("Third");

set.add("Fourth");

set.add("Fifth");

// Adding duplicate elements that will be ignored.

set.add("First");

set.add("Third");

// Adding of null elements.

set.add(null);

set.add(null); // Ignored.

System.out.println("Unordered and No Duplicate HashSet Elements");

System.out.println(set);

}

}

package hashSetTest;

import java.util.ArrayList;

import java.util.HashSet;

public class HashSetExample2 {

public static void main(String[] args)

{

// Create an ArrayList object.

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

al.add("Monday");

al.add("Tuesday");

al.add("Wednesday");

al.add("Thursday");

al.add("Friday");

// Adding duplicate elements.

al.add("Monday");

al.add("Friday");

System.out.println("Original Elements Order ");

System.out.println(al);

// Create HashSet object.

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

// Call addAll() method for adding all elements from existing collection to HashSet.

hset.addAll(al);

System.out.println("Unordered HashSet Elements without Duplicate elements");

System.out.println(hset);

}

}

3. Let’s take an example program where we will see how to remove a specific element from a HashSet and remove all elements available in a set.

**Program source code 3:**

import java.util.HashSet;

public class HashSetExample3 {

public static void main(String[] args)

{

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

hset.add(5);

hset.add(10);

hset.add(15);

hset.add(20);

System.out.println("Initial list of elements");

System.out.println(hset);

// Removing a specific element from HashSet.

hset.remove(10);

System.out.println("List of elements after removing 10");

System.out.println(hset);

HashSet<Integer> hset2 = new HashSet<Integer>();

hset2.add(10);

hset2.add(25);

hset.addAll(hset2);

System.out.println("List of Elements after adding elements from existing collection");

System.out.println(hset);

// Removing all new elements from HashSet.

hset.removeAll(hset2);

System.out.println("List of Elements after removing elements from hset2");

System.out.println(hset);

// Removing all elements available in HashSet.

clear();

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

}

}

4. Let’s create a program where we will check the number of elements in Java HashSet. We will also verify that HashSet is empty or not. Look at the source code to understand better.

**Program source code 4:**

import java.util.HashSet;

import java.util.Set;

public class HashSetExample4 {

public static void main(String[] args)

{

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

// Check that HashSet is empty or not.

System.out.println("Is popularCountries set empty? : " + pCountry.isEmpty());

System.out.println("Number of countries in HashSet before adding: " +pCountry.size());

pCountry.add("INDIA");

pCountry.add("USA");

pCountry.add("UK");

pCountry.add("FRANCE");

// Find size of HashSet.

System.out.println("Number of countries in HashSet after adding: " + pCountry.size());

}

}

**How to create User-defined Object of HashSet?**

Let’s create a program where we will create a user-defined object of HashSet in Java. Look at the source code to understand better.

**Program source code 6:**

package hashSetTest;

public class Student

{

// Declare instance variables.

String name, sName;

int id;

public Student(String name, String sName, int id)

{

this.name = name;

this.sName = sName;

this.id = id;

}

}

package hashSetTest;

import java.util.HashSet;

public class HashSetExample6

{

public static void main(String[] args)

{

// Create a user-defined HashSet object of type Student.

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

// Create objects of Student class and pass the parameters to their constructors.

Student s1 = new Student("John", "RSVM", 0012);

Student s2 = new Student("Shubh", "DPS", 1234);

Student s3 = new Student("Ricky", "DAV", 9876);

// Adding elements to HashSet and pass reference variables s1, s2, s3.

hset.add(s1);

hset.add(s2);

hset.add(s3);

// Traversing HashSet.

for(Student s:hset)

{

System.out.println(s.name+" "+s.sName+" "+s.id);

}

}

}

**When to use HashSet in Java?**

HashSet in Java is used when

1. We don’t want to store duplicate elements.
2. We want to remove duplicate elements from the list.
3. HashSet is more preferred when add and remove operations are more as compared to get operations.
4. We are not working in a multithreading environment.

# LinkedHashSet in Java | Example Program

**LinkedHashSet in Java** is a concrete class that implements [set interface](https://www.scientecheasy.com/2020/10/java-set.html/) and extends [HashSet class](https://www.scientecheasy.com/2020/10/java-hashset.html/) with a doubly linked list implementation.

Java LinkedHashSet class is the same as HashSet class, except that it maintains the ordering of elements in the set in which they are inserted.

In simple words, elements in the HashSet are not ordered, but elements in the LinkedHashSet can be retrieved in the same order in which they were inserted into the set.

**Features of LinkedHashSet**

1. Java LinkedHashSet class contains unique elements like HashSet. It does not allow to insert of duplicate elements. If we try to add a duplicate element, it will fail and the iteration order of the set is not modified.
2. LinkedHashSet class permits to insert null element.
3. LinkedHashSet class in Java is non-synchronized. That means it is not thread-safe.
4. LinkedHashSet class preserve the insertion order of elements
5. It is slightly slower than HashSet.
6. Linked hash set is very efficient for insertion and deletion of elements.

**Java LinkedHashSet Example Programs**

1. **Adding elements:** Let’s create a program where we will perform operations such as adding, checking the size of LinkedHashSet, etc. Look at the program source code to understand better.

import java.util.LinkedHashSet;

public class AddTest

{

public static void main(String[] args)

{

// Create a Linked hash set of generic type.

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

// Checking the size of LinkedHashSet before adding elements.

int size = lhset.size();

System.out.println("Size of LinkedHashSet before adding elements: " +size);

// Adding elements in the linked hash set.

lhset.add("Red"); // lhset.size() is 1.

lhset.add("Green"); // lhset.size() is 2.

lhset.add("Yellow"); // lhset.size() is 3.

lhset.add("Blue"); // lhset.size() is 4.

lhset.add("Orange"); // lhset.size() is 5.

System.out.println("Elements in Set: " +lhset);

int size2 = lhset.size();

System.out.println("Size of LinkedHashSet after adding elements: " +size2);

// Adding duplicate elements that already exist in set.

lhset.add("Red"); // lhset.size() is still 5.

lhset.add("Yellow"); // lhset.size() is still 5.

// Create another set of String type.

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

lhset2.add("Brown");

lhset2.add(null);

// Adding elements of set2 into set.

lhset.addAll(lhset2);

System.out.println("Elements in Set after adding: " +lhset);

}

}

2. **Removing element:** Let’s create another program where we will remove an element from the linked hash set.

**Program source code 2:**

import java.util.LinkedHashSet;

public class RemoveDemo

{

public static void main(String[] args)

{

// Create a Linked hash set of generic type.

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

// Adding elements in the linked hash set.

set.add("A");

set.add("G");

set.add("Y");

set.add("B");

set.add("O");

set.add(null);

System.out.println("Elements in set: " +set);

// Remove a string element from linked hash set.

set.remove(null);

System.out.println("Elements in set after removing: " +set);

System.out.println(set.size()+ " elements in set");

// Create another linked hash set of String type.

LinkedHashSet<String> set2 = new LinkedHashSet<String>();

set2.add("S");

set2.add(null);

System.out.println("Elements in set2: " +set2);

System.out.println(set2.size()+ " elements in set2");

System.out.println("Is S in set2? " +set2.contains("S"));

set.addAll(set2);

System.out.println("Elements in set after adding: " +set);

set.removeAll(set2);

System.out.println("Elements in set after removing set2: " +set);

set.retainAll(set2);

System.out.println("After removing common elements in set2 " + "from set, set is " + set);

}

}

3. **Removing duplicate elements:**Let’s make a program where we will remove duplicate numbers from ArrayList using LinkedHashSet.

**Program source code 3:**

import java.util.ArrayList;

import java.util.LinkedHashSet;

public class RemovingDuplicate {

public static void main(String[] args)

{

int[] num = {20, 30, 50, 30, 40, 80, 10, 10};

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

// Adding numbers to the array list.

for(int i = 0; i < num.length; i++) {

ar.add(num[i]);

}

System.out.println("Original list: " +ar);

LinkedHashSet<Integer> lhset = new LinkedHashSet<>(ar);

System.out.println("New list after removing dupliacte numbers: " +lhset);

}

}

5. **Adding custom objects:** Let’s take an example program where we will add custom objects of type Student into LinkedHashSet and iterate it. Look at the following source code.

**Program source code 5:**

public class Student {

String name;

int id;

String city;

Student(String name, int id, String city){

this.name = name;

this.id = id;

this.city = city;

}

}

import java.util.LinkedHashSet;

public class StudentInfo {

public static void main(String[] args)

{

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

// Creating objects of Students.

Student st1 = new Student("John", 2345, "New York");

Student st2 = new Student("Deep", 1234, "Dhanbad");

Student st3 = new Student("Ricky", 7583, "Cape Town");

// Adding elements (object references) into LinkedHashSet.

lhset.add(st1);

lhset.add(st2);

lhset.add(st3);

// Traversing linked hash set.

for(Student s:lhset){

System.out.println("Name: " +s.name+" "+ "Id: " +s.id+" "+"City: "+s.city);

}

}

}

**When to use LinkedHashSet in Java?**

LinkedHashSet can be used when you do not want duplicate elements (i.e. want to remove duplicate elements) and want to maintain order in which elements are inserted.

If you want to impose different orders such as increasing or decreasing order, you can use TreeSet class

**Which is better to use: HashSet or LinkedHashSet?**

If you do not require to maintain order in which elements are inserted then use HashSet that is more fast and efficient than LinkedHashSet.

# TreeSet in Java | Methods, Example

It sorts elements in ascending order while HashSet does not maintain any order.

**Features of TreeSet class in Java**

1. Java TreeSet contains unique elements similar to the HashSet. It does not allow the addition of a duplicate element.
2. The access and retrieval times are quite fast.
3. TreeSet does not allow inserting null element.
4. TreeSet class is non-synchronized. That means it is not thread-safe.
5. TreeSet maintains the ascending order. When we add elements into the collection in any order, the values are automatically presented in sorted, ascending order.
6. Java TreeSet internally uses a TreeMap for storing elements.

**Important TreeSet Methods in Java**

**1. boolean add(Object o):** This method is used to add the specified element to this set if it is not already present.

**2. boolean addAll(Collection c):** This method is used to add all the elements of the specified collection to the set.

**3. void clear():** It is used to remove all the elements from the set.

**4. boolean isEmpty():** This method is used to check that the set has elements or not. It returns true if the set contains no elements otherwise returns false.

**5. boolean remove(Object o):** This method is used to remove the specified element from the set if it is present.

**6. int size():** This method is used to get the total number of elements in the set.

### Important Methods Defined by SortedSet Interface

reeSet implements SortedSet interface, all the methods defined by SortedSet interface can be used while using TreeSet class. They are as:

**1. Object first():** It is used to get the first (lowest) element currently in the sorted set.

**2. Object last():** This method returns the last (highest) element currently in the sorted set.

**4. SortedSet headSet(Object toObject):** This method returns the collection of elements that are less than the specified element.

**5. SortedSet subSet(Object fromElement, Object toElement):** It returns elements from the set that lie between the given range in which fromElement is included and toElement is excluded.

**6. SortedSet tailSet(Object fromElement):** It returns elements from the set that is greater than or equal to the specified element.

**Java TreeSet Example Programs**

import java.util.Set;

import java.util.TreeSet;

public class TreeSetEx1 {

public static void main(String[] args)

{

// Create a tree set.

Set<String> ts = new TreeSet<>();

// Check Set is empty or not.

boolean empty = ts.isEmpty();

System.out.println("Is TreeSet empty: " +empty);

// Checking the size of TreeSet before adding elements into it.

int size = ts.size();

System.out.println("Size of TreeSet: " +size);

// Adding elements into TreeSet.

ts.add("India"); // ts.size() is 1.

ts.add("USA"); // ts.size() is 2.

ts.add("Australia"); // ts.size() is 3.

ts.add("New zealand"); // ts.size() is 4.

ts.add("Switzerland"); // ts.size() is 5.

System.out.println("Sorted TreeSet: " +ts);

int size2 = ts.size();

System.out.println("Size of TreeSet after adding elements: " +size2);

}

}

import java.util.TreeSet;

public class TreeSetEx2

{

public static void main(String[] args)

{

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

// Add Strings to tree set.

ts.add("India");

ts.add("USA");

ts.add("Australia");

ts.add("New zealand");

ts.add("Switzerland");

// Checking for a specific element in set.

boolean element = ts.contains("USA");

System.out.println("Is USA in TreeSet: " +element);

// Removing element from the tree set.

ts.remove("New zealand");

System.out.println("Sorted tree set: " +ts);

ts.clear();

System.out.println("Elements in tree set: " +ts);

}

}

Program source code 4:

import java.util.TreeSet;

public class TreeSetEx4

{

public static void main(String[] args)

{

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

ts.add(25);

ts.add(80);

ts.add(05);

ts.add(100);

ts.add(90);

ts.add(200);

ts.add(300);

System.out.println("Sorted TreeSet: " +ts);

// Using methods of NavigableSet interface.

System.out.println("Largest element less than 100: " +ts.lower(100));

System.out.println("Smallest element greater than 100: " +ts.higher(100));

System.out.println("Floor: " +ts.floor(85));

System.out.println("Ceiling: " +ts.ceiling(10));

System.out.println(ts.pollFirst()); // Remove and retrieve the first element from the set.

System.out.println(ts.pollLast()); // Remove and retrieve the last element from the set.

System.out.println("New Treeset: " +ts);

System.out.println("HeadSet: " +ts.headSet(90,true));

System.out.println("SubSet: " +ts.subSet(90, true, 200, true));

}

}

## When to Use TreeSet in Java?

TreeSet can be used when we want unique elements in sorted order.

### Which is better to use: HashSet or TreeSet?

If you want to store unique elements in sorted order then use TreeSet, otherwise, use HashSet with no ordering of elements. This is because HashSet is much faster than TreeSet.

# Map in Java | Map.Entry, Example

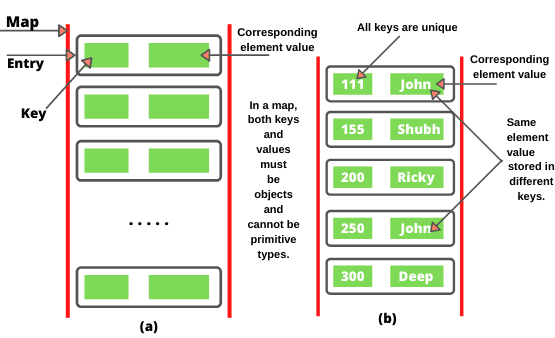
A **map in Java** is a container object that stores elements in the form of key and value pairs. A key is a unique element (object) that serves as an “index” in the map.

The element that is associated with a key is called value. A map stores the values associated with keys. In a map, both keys and values must be objects and cannot be primitive types.

A map cannot have duplicate keys. Each key maps to only one value. This type of mapping is called **one-to-one mapping in java**.

All keys must be unique, but values may be duplicate (i.e. the same value can be stored to several different keys).

A key and its associated value are called an entry that is stored in a map as shown in the below figure. After the entry (key/value pairs) is stored in a map, we can retrieve (get) its value by using its key.

****

**Java Map.Entry Interface**

The **Map.Entry interface** enables us to work on an entry in the map. An entry of a map is an object of type Map.Entry interface, where Entry is an inner interface of Map interface.

methods defined by Map.Entry interface in Java. They are as follows:

**1. boolean equals(Object obj):** It is used to check for equality the specified object with the other existing object. It returns true if the specified object obj is a Map.Entry whose key and value are equal to that of the existing object.

**2. K getKey():** It is used to retrieve the key for a map entry. Its return type is key.

**3. V getValue():** It is used to get the value for a map entry. Its return type is value.

**4. int hashCode():** It returns hash code value for a map entry.

**5. void setValue(V value):** This method is used to replace the existing value corresponding to this entry with the specified value and returns the replaced value.

**Map Methods in Java**

**1. V put(K key, V value):** It is used to add an entry with specified key and value in the map.

**2. void putAll(Map m):** It is used to add all entries from into this map.

**3. V putIfAbsent(K key, V value):** It is used to add specified value with specified key in the map only if it is not already specified.

**4. V remove(Object key):** This method is used to delete an entry for the specified key. It will return null if the key is not in the map.

**5. boolean remove(Object key, Object value):** This method is used to remove the specified value associated with specified key from the map.

**6. Set<K> keySet():** This method returns a set consisting of the keys in the invoking map. It provides a set-view of the keys.

**7. void clear():** This method is used to remove all entries from the map.

**8. V get(Object key):** This method returns the value for the specified key in this map.

9. **boolean isEmpty():** This method is used to check whether the map contains any entries. It returns true if the invoking map is empty, otherwise returns false if it contains at least one key.

**11. int size():** The size() method returns the number of entries (number of key/value pairs) in the map.

**12. V replace(K key, V value):** This method is used to replace the specified value for a specified key.

**13. boolean replace(K key, V oldValue, V newValue):** This method is used to replace old value with new value for a specified key.

**Java Map Example Programs**

**1. Adding entries:** Let’s create a program where we will add entries in map using put() method. We will also check a map is empty or not before adding entries in a map. Look at the source code to understand better.

**Program source code 1:**

import java.util.HashMap;

import java.util.Map;

public class AddDemo

{

public static void main(String[] args)

{

// Create a map of generic type.

Map<Integer, String> map = new HashMap<>();

// Checking map is empty or not.

boolean isEmpty = map.isEmpty();

System.out.println(" Is Map is empty? " +isEmpty);

// Adding entries in the map. Call put() method to add entries in map.

map.put(101, "Red");

map.put(103, "Green");

map.put(102, "Yellow");

map.put(104, "Blue");

map.put(106, "Pink");

System.out.println("Entries in Map: " +map);

int size = map.size();

System.out.println("No. of entries in Map: " +size);

// Create another map.

Map<Integer,String> map2 = new HashMap<>();

map2.put(115, "Brown");

map2.put(120, "Purple");

map2.put(125, "Green");

map.putAll(map2);

System.out.println("Entries in updated Map: " +map);

}

}

**2. Removing entries:** Let’s create a program where we will remove entry from a map using remove() method.

**Program source code 2:**

import java.util.HashMap;

import java.util.Map;

public class AddDemo

{

public static void main(String[] args)

{

// Create a map of generic type.

Map<Integer, String> map = new HashMap<>();

// Adding entries in the map.

map.put(101, "Red");

map.put(103, "Green");

map.put(102, "Yellow");

map.put(104, "Blue");

map.put(106, "Pink");

// Removing an entry for the specified key.

map.remove(104);

System.out.println("Entries in Map after removing an entry: " +map);

// Removing specified value associated with the specified key from the map.

map.remove(106,"Pink");

System.out.println("Entries in Map after removing Pink: " +map);

map.clear();

System.out.println(map);

}

}

1. **Replacing value:** Let’s make a program where we will replace a specified value for a specified key. Look at the source code

import java.util.HashMap;

import java.util.Map;

public class ReplaceDemo

{

public static void main(String[] args)

{

Map<String, String> map = new HashMap<>();

map.put("A", "Apple");

map.put("B", "Boy");

map.put("C", "Cat");

map.put("D", "Dog");

map.put("E", "Elephant");

System.out.println("Orignal Entries in Map: " +map);

// Replacing a specified value for a specified key.

map.replace("E", "Element");

System.out.println("Updated Entries in Map after replacing: " +map);

// Replace old value with a new value.

map.replace("B", "Boy", "Bucket");

1. System.out.println(map);

}

}

1. **4. Getting keys and values:** Let’s take an example program where we will get a set-view of keys and values of the invoking map.
2. **Program source code 4:**

import java.util.Collection;

import java.util.HashMap;

import java.util.Map;

import java.util.Set;

public class KeySetDemo

{

public static void main(String[] args)

{

Map<String, String> map = new HashMap<>();

map.put("B", "B");

map.put("D", "H");

map.put("A", "A");

map.put("C", "R");

map.put("E", "T");

Set<String> keys = map.keySet();

System.out.println("Set view of keys: " +keys);

Collection<String> values = map.values();

System.out.println("Collection view of values: " +values);

}

}

**5. Getting value for key:** Let’s create a program where we will get a value for the specified key in the map. We will also check whether the map contains an entry for the specified key.

**Program source code 5:**

import java.util.HashMap;

import java.util.Map;

public class GettingValueDemo

{

public static void main(String[] args)

{

Map<String, String> map = new HashMap<>();

map.put("V", "Violet");

map.put("I", "Indigo");

map.put("B", "Blue");

map.put("G", "Green");

map.put("Y", "Yellow");

map.put("O", "Orange");

map.put("R", "Red");

String value = map.get("V"); // It will return a value for specified key in this map.

System.out.println(value);

boolean entryKey = map.containsKey("B");

System.out.println(entryKey);

boolean entryValue = map.containsValue("Brown");

System.out.println(entryValue);

}

}

**Iterating Map in Java using Iterator over Map.Entry<K, V>**

An iterator is an interface that contains methods to retrieve the entries of a map one by one. It provides three methods:

* **boolean hasNext ()**: It returns true if the iterator has more elements.
* **element next ()**: It returns the next element in the iterator.
* **void remove ()**: It removes the last element returned by the iterator.

Note that we cannot iterate over a map directly using iterator because Map interface is not the part of the Collection interface. To iterate map using iterator, you must familiar with Map.Entry<K, V> interface.

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import java.util.Map.Entry;

public class IterationTest1

{

public static void main(String[] args)

{

Map<String, String> map = new HashMap<>(); // Creating a map.

map.put("V ", " Violet");

map.put("I ", " Indigo");

map.put("B ", " Blue");

map.put("G ", " Green");

map.put("Y ", " Yellow");

map.put("O ", " Orange");

map.put("R ", " Red");

Iterator<Entry<String, String>> itr = map.entrySet().iterator(); // entrySet is a method that is used to get view of entries of a map.

System.out.println("Iterating Entries of Map");

while(itr.hasNext())

{

Object key = itr.next();

System.out.println(key);

}

Iterator<String> itr2 = map.keySet().iterator(); // keySet is a method that is used to get view of keys of a map.

System.out.println("Iterating Keys of Map");

while(itr2.hasNext())

{

Object keyView = itr2.next();

System.out.println(keyView);

}

Iterator<String> itr3 = map.values().iterator(); // values is a method that is used to get values of keys of a map.

System.out.println("Iterating Values of Map");

while(itr3.hasNext())

{

Object valuesView = itr3.next();

System.out.println(valuesView);

}

}

}

………………………………………………………………………………

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import java.util.Map.Entry;

public class IterationTest2

{

public static void main(String[] args)

{

Map<String, String> map = new HashMap<>();

map.put("V ", " Violet");

map.put("I ", " Indigo");

map.put("B ", " Blue");

map.put("G ", " Green");

map.put("Y ", " Yellow");

map.put("O ", " Orange");

map.put("R ", " Red");

Iterator<Entry<String, String>> itr = map.entrySet().iterator(); // entrySet is a method that is used to get view of entries of a map.

System.out.println("Iterating Entries of Map");

while(itr.hasNext())

{

Object key = itr.next();

System.out.println(key);

}

// Removing last entry returned by Iterator

itr.remove(); // This method will remove last entry of a map while iterating.

System.out.println("Entries of Map after removing: " +map.entrySet());

}

}

**……….**

**Iterating over keys or values using keySet() and value() methods**

This technique is useful when you want to get a set view of keys or values of a map. Using keySet(), values(), and for-each loop, you can iterate over keys or values of a map.

import java.util.HashMap;

import java.util.Map;

public class IterationTest3

{

public static void main(String[] args)

{

Map<Integer, String> map = new HashMap<>();

map.put(101, " John");

map.put(202, " Ricky");

map.put(303, " Deep");

map.put(404, " Mark");

map.put(505, " Maya");

for (Integer rollNo : map.keySet()) // Iterating over keys of a map using keySet() method.

System.out.println("Roll No: " + rollNo);

System.out.println();

for (String name : map.values()) // Iterating over values of a map using values() method.

System.out.println("Name: " + name);

}

}

**Iterating Map using Map.Entry<K,V>method**

Map.Entry<K,V> is an interface that is used to work on an entry in the map. It returns a collection view of the map. Each Map.Entry object contains one key/value pair.

1. **getKey()**: It is used to retrieve the key for a map entry. Its return type is key.  
2. **getValue()**: It is used to get the value for a map entry. Its return type is value.  
3. **entrySet()**: It returns a set view of entries of a map.

**Program source code 4:**

import java.util.HashMap;

import java.util.Map;

public class IterationTest4

{

public static void main(String[] args)

{

Map<Integer, String> map = new HashMap<>();

map.put(101, " John");

map.put(202, " Ricky");

map.put(303, " Deep");

map.put(404, " Mark");

map.put(505, " Maya");

for (Map.Entry<Integer,String> entry : map.entrySet()) // Iterating over entries of a map using entrySet() method.

{

System.out.println("Roll No: " + entry.getKey() + ", Name: " + entry.getValue());

}

}

}

# HashMap in Java | Methods, Use, Example

**HashMap in Java** is an unordered collection that stores elements (objects) in the form of key-value pairs (called entries).

It is expressed as HashMap<Key, Value>, or HashMap<K, V>, where K stands for key and V for value. Both Key and value are objects. HashMap uses an object to retrieve another object.

If the key is provided, its associated value can be easily retrieved from the HashMap. Keys in the map must be unique which means we cannot use duplicate data for keys in the HashMap.

**Features of Java HashMap class**

1. The underlying data structure of HashMap is [HashTable](https://www.scientecheasy.com/2020/12/hashtable-in-java.html/). In simple words, HashMap internally uses hash table for storing entries. That means accessing and adding an entry almost as fast as accessing an array.

2. In HashMap, insertion order is not preserved (i.e. maintains no order). Which means we cannot retrieve keys and values in the same order in which they have been inserted into the HashMap.

3. It is based on the Hashcode of keys, not on the hash code of values.

4. Java HashMap contains only unique keys that means no duplicate keys are allowed but values can be duplicated. We retrieve values based on the key.

5. Heterogeneous objects are allowed for both keys and values.

6. Java HashMap can have only one null key because duplicate keys are not allowed.

7. Multiple null values are allowed in the HashMap.

8. HashMap in Java is not synchronized that means while using multiple threads on the HashMap object, we will get unreliable results.

9. Java HashMap implements Cloneable, and Serializable interfaces but not implements Random Access.

10. HashMap is the best choice if our frequent operation is a search operation.

11. If no element exists in the HashMap, it will throw an exception named **NoSuchElementException**.

**Constructors of HashMap class**

1. **HashMap():** It is used to construct an empty HashMap object with the default initial capacity of 16 and the default fill ratio (load factor) is 0.75. The syntax to create a hash map object is as follows:
2. HashMap hmap = new HashMap();
3. HashMap<K, V> hmap = new HashMap<K,V>(); // Generic form.
4. **2. HashMap(int initialCapacity):** It is used to create an empty hash map object with a specified initial capacity under the default load factor 0.75. The general syntax is as follows:
5. HashMap<K,V> hmap = new HashMap<K,V>(int initialCapacity);
6. **HashMap(Map m):** This constructor is used to create hash map object by initializing the elements of the given Map object m.

**HashMap Methods in Java**

1. **void clear():** It is used to remove entries from the specified map.

2. **boolean isEmpty():** This method is used to check whether the map is empty or not. If there are no key-value mapping present in the hash map then it returns true, else false.

3. **Object clone():** It is used to create a shallow copy of this HashMap. Only hashmap and all entries are cloned, not elements. Both this map and new map share references to the same keys and values.

4. **Set entrySet():** It is used to return a set view containing all of the key/value pairs in this map.

5. **Set keySet():** This method is used to retrieve a set view of the keys in this map.

6. **V put(Object key, Object value):** This method is used to insert an entry in the map.

7. **void putAll(Map map):** This method is used to insert all the entries of the specified map to another map.

8. **V putIfAbsent(K key, V value):** This method adds the specified value associated with the specified key in the map only if it is not already specified.

9. **V remove(Object key):** This method is used to delete an entry for the specified key.

10. **boolean remove(Object key, Object value):** This method removes the specified value associated with specific key from the map.

11. **int size():** This method returns the number of entries in the map.

12. **Collection values():** This method returns a collection view containing all of the values in the map.

13. **V get(Object key):** This method is used to retrieve the value associated with a key. Its return type is Object.

14. **V replace(K key, V value):** This method is used to replace the specified value for a specified key.

15. **boolean replace(K key, V oldValue, V newValue):** This method is used to replace the old value with the new value for a specified key.

**Java HashMap Example Programs**

import java.util.HashMap;

public class HashMapEx1 {

public static void main(String[] args)

{

// Create a HashMap.

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

// Checking HashMap is empty or not.

boolean empty = hmap.isEmpty();

System.out.println("Is HashMap empty: " +empty);

// Adding entries in the hash map.

hmap.put("John", 24); // hmap.size() is 1.

hmap.put("Deep", 22); // hmap.size() is 2.

hmap.put("Shubh", 15); // hmap.size() is 3.

hmap.put("Riky", 22); // hmap.size() is 4. // Adding duplicate value.

hmap.put("Mark", 30); // hmap.size() is 5.

System.out.println("Entries in HashMap: " +hmap);

int size = hmap.size();

System.out.println("Size of HashMap: " +size);

// Adding null key and value.

hmap.put(null, null); // hmap.size() is 6.

System.out.println("Updated entries in HashMap: " +hmap);

}

}

import java.util.HashMap;

public class HashMapEx2 {

public static void main(String[] args)

{

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

hmap.put(5, "Banana");

hmap.put(10, "Mango");

hmap.put(15, "Apple");

System.out.println("Entries in HashMap: " +hmap);

System.out.println("Size of HashMap: " +hmap.size());

// Adding duplicate key in hash map.

hmap.put(10, "Guava"); // Still hmap.size is 3.

hmap.put(20, "Banana"); // Adding duplicate value.

System.out.println("Updated entries in HashMap: " +hmap);

System.out.println("Size after adding duplicate value: " +hmap.size());

}

}

**…………….**

import java.util.HashMap;

public class HashMapEx3 {

public static void main(String[] args)

{

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

hmap.put('R', "Red");

hmap.put('O', "Orange");

hmap.put('G', "Green");

hmap.put('B', "Brown");

hmap.put('W', "White");

// Displaying HashMap entries.

System.out.println("Entries in HashMap: " +hmap);

// Removing Key-Value pairs for key 'B'.

Object removeEntry = hmap.remove('B');

System.out.println("Removed Entry: " +removeEntry);

System.out.println("HashMap Entries after remove: " +hmap);

// Checking entry is removed or not.

Object removeEntry2 = hmap.remove('W', "White");

System.out.println("Is entry removed: " +removeEntry2);

System.out.println("Updated HashMap entries: " +hmap);

}

}

4. Let’s take an example program where we will replace a specified value for the specified key. Look at the source code.

import java.util.HashMap;

public class HashMapEx4 {

public static void main(String[] args)

{

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

hmap.put('R', "Red");

hmap.put('O', "Orange");

hmap.put('G', "Green");

hmap.put('B', "Brown");

hmap.put('W', "White");

// Displaying HashMap entries.

System.out.println("Entries in HashMap: " +hmap);

// Replacing specified value for the specified key.

Object replaceValue = hmap.replace('B', "Black");

System.out.println("Replaced value: " +replaceValue);

System.out.println("Updated entries in HashMap: " +hmap);

boolean replaceValue2 = hmap.replace('G', "Green", "Greenish");

System.out.println("Is value replaced: " +replaceValue2);

System.out.println(hmap);

}

}

# What is Hashtable in Java with Example

**Hashtable class in Java** is a concrete implementation of abstract Dictionary class.

It is a data structure similar to Java HashMap that can store a collection of elements (objects) in the form of key-value pairs (entries).

Key objects must implement hashCode() and equals() methods to store and retrieve values from the Hashtable.

In other words, Hashtable can only store key objects that override hashCode() and equals() methods defined by the Object class.

he main difference between Hashtable and HashMap is the way they work with thread access.

* Hashtable class is a synchronized class that means it is thread-safe. Multiple threads cannot access the same instance of the Hashtable class concurrently (at the same time).
* HashMap class is not synchronized which means it is not thread-safe. Multiple threads can access the same instance of HashMap class simultaneously. Therefore, it is safe to use only when one thread uses an object.
* **Features of Hashtable**

There are several features of Hashtable in Java that must keep in mind to use it.

1. The underlying data structure for Java Hashtable is a hash table only.

2. Insertion order is not preserved. That means it does not maintain insertion order.

3. Duplicate keys are not allowed but values can be duplicated.

4. Heterogeneous objects are allowed for both keys and values.

5. Null is not allowed for both key and values. If we attempt to store null key or value, we will get a RuntimeException named NullPointerException.

6. Java Hashtable implements Serializable and Cloneable interfaces but not random access.

7. Every method present in Hashtable is synchronized. Hence, Hashtable object is thread-safe.

8. Hashtable is the best choice if our frequent operation is a retrieval (search) operation.

9. Since Hashtable is synchronized, its operations are slower as compared to HashMap in java.

**Hashtable Methods in Java**

**1. void clear():** It is used to clear all key-value pairs from the Hashtable.

**2. Object clone():** This method is used to create a shallow copy of this hashtable.

**3. V put(K key, V value):** This method is used to map key-value pairs into the Hashtable.

**4. void putAll(Map t):** This method copies all of the key-value pairs (mappings) from the specified map into the Hashtable.

**5. boolean isEmpty():** It returns true if there are no entries in the Hashtable.

**6. int size():** This method is used to retrieve the number of key-value pairs in the Hashtable.

**7. V remove(Object key):** This method is used to remove the key (and its associated value) from the hashtable.

**8. boolean remove(Object key, Object value):** This method removes the entry for the specified key only if it is currently mapped to the specified value.

**9. V replace(K key, V value):** This method is used to replace the entry for the specified key only if it is currently mapped to some value.

**10. boolean replace(K key, V oldValue, V newValue):** This method replaces the key-value pair for the specified key only if currently mapped to the specified value.

**Java Hashtable Example Programs**

1. Let’s create a program where we will perform various operations such as adding, removing, checking hash table is empty or not before adding elements, and size.

import java.util.Hashtable;

public class HashtableEx {

public static void main(String[] args)

{

// Create a Hashtable object.

Hashtable<Integer, String> ht = new Hashtable<Integer, String>();

// Checking hashtable is empty or not.

boolean isEmpty = ht.isEmpty();

System.out.println("Is hash table empty: " +isEmpty);

// Adding entries using put() method in hash table.

ht.put(1, "One"); // ht.size() is 1.

ht.put(2, "Two"); // ht.size() is 2.

ht.put(3, "Three"); // ht.size() is 3.

ht.put(4, "Four"); // ht.size() is 4.

ht.put(5, "Five"); // ht.size() is 5.

ht.put(6, "Six"); // ht.size() is 6.

System.out.println("Displaying entries in hash table: " +ht);

int size = ht.size();

System.out.println("Size of hash table: " +size);

// Removing last entry.

String removeE = ht.remove(6);

System.out.println("Removed entry: " +removeE);

System.out.println("Updated entries in hash table: " +ht);

// Getting the value of 4.

String getValue = ht.get(4);

System.out.println("Getting the value of 4: " +getValue);

System.out.println("Getting the value of 2: " +ht.get(2));

}

}

……………..

import java.util.Hashtable;

public class HashtableEx2 {

public static void main(String[] args)

{

// Create a Hashtable object.

Hashtable<String, Integer> ht = new Hashtable<>();

ht.put("John", 20);

ht.put("Shubh", 30);

ht.put("Peter", 25);

ht.put("Deep", 15);

ht.put("Jonshan", 40);

System.out.println("Original entries of hash table: " +ht);

// Replacing an entry for specified key from hash table.

Integer replace = ht.replace("Peter", 60);

System.out.println("Replacing entry for specified key: " +replace);

System.out.println("Updated entries in hash table: " +ht);

// Checking specified key present in hash table.

boolean containsKey = ht.containsKey("Shubh");

System.out.println("Is key Shubh in hash table: " +containsKey);

// Checking specified value present in hash table.

boolean containsValue = ht.containsValue(40);

System.out.println("Is value 40 in hash table: " +containsValue);

}

}

# LinkedHashMap in Java | Methods, Example

**LinkedHashMap in Java** is a concrete class that is HashTable and LinkedList implementation of Map interface. It stores entries using a doubly-linked list.

Java LinkedHashMap class extends the HashMap class with a linked-list implementation that supports an ordering of the entries in the map.

insertion order is not preserved in the HashMap because it is based on the hashCode of Key. But in the case of LinkedHashMap, the insertion order of elements is preserved because it is based on the Key insertion order, that is, the order in which keys are inserted in the map

**Features of LinkedHashMap in Java**

1. The underlying data structure of LinkedHashMap is HashTable and [LinkedList](https://www.scientecheasy.com/2020/09/java-linkedlist.html/).

2. Java LinkedHashMap maintains the insertion order. The entries in Java LinkedHashMap can be retrieved either in the order in which they were inserted into the map (known as insertion order) or in the order in which they were last accessed, from least to most recently accessed.

3. LinkedHashMap contains unique elements. It contains values based on keys.

4. LinkedHashMap allows only one null key but can have multiple null values.

5. LinkedHashMap in Java is non synchronized. That is, multiple [threads](https://www.scientecheasy.com/2020/08/java-thread-tutorial.html/) can access the same LinkedHashMap object simultaneously.

**LinkedHashMap Methods in Java**

The methods of LinkedHashMap are exactly the same as HashMap class methods, except for one method that is added by LinkedHashMap. This method is removeEldestEntry().

The general syntax for this method is as follows:

protected boolean removeEldestEntry(Map.Entry<K,V> e)

The parameter e is the least recently added entry in the map, or if it is an access-ordered map, the least recently accessed entry.

This method returns true if the map removes this eldest (oldest) entry. If this entry is retained, or not removed, this method returns false.

**Java LinkedHashMap Example Programs**

import java.util.LinkedHashMap;

public class LinkedHashMapEx1 {

public static void main(String[] args)

{

// Create a LinkedHashMap instance.

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

// Checking the size of linked hash map before adding entries.

int size = lhmap.size();

System.out.println("Size of LinkedHashMap before adding entries: " +size);

// Checking linked hash map is empty or not before adding entries.

boolean isEmpty = lhmap.isEmpty();

System.out.println("Is LinkedHashMap empty: " +isEmpty);

// Adding entries in linked hash map.

lhmap.put("John", 30);

lhmap.put("Peter", 25);

lhmap.put("Ricky", 23);

lhmap.put("Deep", 28);

lhmap.put("Mark", 32);

System.out.println("Display entries in LinkedHashMap");

System.out.println(lhmap);

int size2 = lhmap.size();

System.out.println("Size of LinkedHashMap after adding entries: " +size2);

// Adding null as key and value.

lhmap.put(null, null);

System.out.println(lhmap);

}

import java.util.LinkedHashMap;

public class LinkedHashMapEx2 {

public static void main(String[] args)

{

LinkedHashMap<String, String> lhmap = new LinkedHashMap<>(16, 0.75f, true);

lhmap.put("En", "English");

lhmap.put("Hi", "Hindi");

lhmap.put("Ta", "Tamil");

lhmap.put("De", "German");

lhmap.put("Fr", "French");

System.out.println("Initially, entries in LinkedHashMap lhmap: " +lhmap);

System.out.println("Value corresponding to key Hi: " +lhmap.get("Hi"));

System.out.println("Value corresponding to key En: " +lhmap.get("En"));

System.out.println("After accessing entries Hi and En: " +lhmap);

System.out.println("\n");

LinkedHashMap<String, String> lhmap2 = new LinkedHashMap<>(16, 0.75f, false);

lhmap2.put("En", "English");

lhmap2.put("Hi", "Hindi");

lhmap2.put("Ta", "Tamil");

lhmap2.put("De", "German");

lhmap2.put("Fr", "French");

System.out.println("Initially, entries in LinkedHashMap lhmap2: " +lhmap2);

System.out.println("Value corresponding to key Hi: " +lhmap.get("Hi"));

System.out.println("Value corresponding to key En: " +lhmap.get("En"));

System.out.println("After accessing entries Hi and En: " +lhmap2);

}

}

………….

import java.util.LinkedHashMap;

public class LinkedHashMapEx3 {

public static void main(String[] args)

{

LinkedHashMap<String, String> lhmap = new LinkedHashMap<>();

lhmap.put("En", "English");

lhmap.put("Hi", "Hindi");

lhmap.put("Ta", "Tamil");

lhmap.put("De", "German");

lhmap.put("Fr", "French");

System.out.println("Entries in LinkedHashMap lhmap: " +lhmap);

// Call remove() method to delete an entry for specified key.

lhmap.remove("De");

System.out.println("Updated Entries in LinkedHashMap: " +lhmap);

// Call replace() method to replace specified value for a specified key.

lhmap.replace("En", "English-US");

System.out.println("After replacing, updated entries in LinkedHashMap: " +lhmap);

// Call containsValue() method to determine specified value for specified key.

boolean value = lhmap.containsValue("Hindi");

System.out.println("Is Hindi present in LinkedHashMap: " +value);

}

}

**Program based on removeEldestEntry() method**

import java.util.LinkedHashMap;

import java.util.Map;

public class LinkedHashMapEx1 {

public static void main(String[] args)

{

final int max = 5;

LinkedHashMap<String, String> lhmap = new LinkedHashMap<String,String>(){

protected boolean removeEldestEntry(Map.Entry<String, String> e)

{

return size() > max;

}

};

lhmap.put("R", "Red");

lhmap.put("G", "Green");

lhmap.put("B", "Brown");

lhmap.put("O", "Orange");

lhmap.put("P", "Pink");

System.out.println("Initial Entries of LinkedHashMap");

System.out.println(lhmap);

// Adding more entry into linked hash map.

lhmap.put("W", "White");

System.out.println("Displaying Map after adding more entry: " +lhmap);

lhmap.put("Y", "Yellow"); // Adding one more entry into linked hash map.

System.out.println("Displaying Map after adding one more entry: " +lhmap);

}

}

**When to use LinkedHashMap in Java?**

LinkedHashMap can be used when you want to preserve the insertion order. Java LinkedHashMap is slower than HashMap but it is suitable when more number of insertions and deletions happen.

**Which implementation is better to use: HashMap or LinkedHashMap?**

Both HashMap and LinkedHashMap classes provide comparable performance but HashMap is a natural choice if the ordering of elements is not an issue.

Adding, removing, and finding entries in a LinkedHashMap is slightly slower than in HashMap because it needs to maintain the order of doubly linked list in addition to the hashed data structure.

# TreeMap in Java | Methods, Example

It provides an efficient way of storing key/value pairs in sorted order automatically and allows rapid retrieval. It was added in JDK 1.2 and present in java.util.TreeMap.

The two main difference between HashMap and TreeMap is that

* HashMap is an unordered collection of elements while TreeMap is sorted in the ascending order of its keys. The keys are sorted either using Comparable interface or Comparator interface.
* HashMap allows only one null key while TreeMap does not allow any null key.

**Important Features of  TreeMap**

There are several important features of TreeMap in Java that should be kept in mind. They are as follows:

1. The underlying data structure of Java TreeMap is a red-black binary search tree.

2. TreeMap contains only unique elements.

3. TreeMap cannot have a null key but can have multiple null values.

4. Java TreeMap is non synchronized. That means it is not thread-safe. We can create a synchronized version of map by calling Collections.synchronizedMap() on the map.

5. TreeMap in Java maintains ascending order. The mappings are sorted in treemap either according to the natural ordering of keys or by a comparator that is provided during the object creation of TreeMap depending upon the constructor used.

**Methods of Java TreeMap class**

1. **void clear():** This method removes all objects (entries) from the tree map.

2. **V put(K key, V value):** This method is used to insert a key/value pair in the tree map.

3. **void putAll(Map m):** It is used to add key/value pairs from Map m to the current tree map.

4. **V remove(Object key):** It is used to remove the key-value pair of the specified key from the tree map.

5. **V get(Object key):** This method is used to retrieve the value associated with key. If key is null, it will throw NullPointerException.

6. **K firstKey():** It is used to retrieve key of first entry in the sorted order from the map. If the tree map is empty, it will throw NoSuchElementException.

7. **K lastKey():** It is used to retrieve key of first entry in the sorted order from the map. If the tree map is empty, it will throw NoSuchElementException.

10. **int size():** This method returns the number of entries (objects) in the tree map.

11. **Set keySet():** This method returns a set (collection) of all keys of the tree map.

12. **Set entrySet():** This method returns a set view containing all key/value pairs in the tree map.

**Java TreeMap Example Programs**

import java.util.HashMap;

import java.util.LinkedHashMap;

import java.util.TreeMap;

public class TreeMapEx1 {

public static void main(String[] args)

{

// Create a HashMap.

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

hmap.put("R", "Red");

hmap.put("G", "Green");

hmap.put("B", "Brown");

hmap.put("O", "Orange");

hmap.put("P", "Pink");

System.out.println("Entries of HashMap: " +hmap);

// Create a TreeMap from the previous HashMap.

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

System.out.println("Entries in ascending order of keys: " +tmap);

// Create a LinkedHashMap.

LinkedHashMap<String, String> lhmap = new LinkedHashMap<>();

lhmap.put("R", "Red");

lhmap.put("G", "Green");

lhmap.put("B", "Brown");

lhmap.put("O", "Orange");

lhmap.put("P", "Pink");

System.out.println("Entries in LinkedHashMap: " +lhmap);

}

}

Let’s create a program and perform add, remove, and replace operations in the map.

import java.util.TreeMap;

public class TreeMapEx2{

public static void main(String[] args)

{

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

int size = tmap.size();

System.out.println("Size of TreeMap before adding entries: " +size);

boolean isEmpty = tmap.isEmpty();

System.out.println("Is TreeMap empty: " +isEmpty);

// Adding entries in tree map.

tmap.put("John", 25);

tmap.put("Ricky", 22);

tmap.put("Deep", 28);

tmap.put("Mark", 20);

tmap.put("Peter", 30);

System.out.println("Entries in ascending order: " +tmap);

tmap.remove("Mark");

System.out.println("Entries of TreeMap after removing: " +tmap);

tmap.replace("Peter", 18);

System.out.println("Updated enrties of TreeMap: " +tmap);

}

}

Let’s take an example program based on entrySet(), keySet(), values(), get(), containsKey(), and containsValue() methods.

import java.util.TreeMap;

public class TreeMapEx1{

public static void main(String[] args)

{

TreeMap<Character, String> tmap = new TreeMap<>();

tmap.put('A', "Apple");

tmap.put('P', "Parot");

tmap.put('C', "Cat");

tmap.put('B', "Boy");

tmap.put('D', "Dog");

Object entrySet = tmap.entrySet();

System.out.println("Entry set: " +entrySet);

System.out.println("Key set: " +tmap.keySet());

System.out.println("Value set: " +tmap.values());

Object vGet = tmap.get('C');

System.out.println("C: " +vGet);

boolean containsKey = tmap.containsKey('B');

System.out.println("Is key 'B' present in map: " +containsKey);

boolean containsValue = tmap.containsValue("Apple");

System.out.println("Is Apple present in map: " +containsValue);

}

}

Let’s take an example program based on ceilingEntry(), ceilingKey(), firstEntry(), lastEntry(), floorEntry() methods.

import java.util.TreeMap;

public class TreeMapEx1{

public static void main(String[] args)

{

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

tmap.put(25, "John");

tmap.put(22, "Shubh");

tmap.put(30, "Ricky");

tmap.put(35, "Peter");

tmap.put(18, "Johnson");

System.out.println("ceilingEntry: " +tmap.ceilingEntry(32));

System.out.println("ceilingKey: " +tmap.ceilingKey(32));

System.out.println("firstEntry: " +tmap.firstEntry());

System.out.println("lastEntry: " +tmap.lastEntry());

System.out.println("floorEntry: " +tmap.floorEntry(31));

System.out.println("HigherEntry: " +tmap.higherEntry(30));

System.out.println("LowerEntry: " +tmap.lowerEntry(30));

System.out.println("pollFirstEntry: " +tmap.pollFirstEntry());

System.out.println("pollLastEntry: " +tmap.pollLastEntry());

}

}

## When to use TreeMap in Java?

TreeMap is slower than HashMap but it is preferred:

* When we do not want null key in the map.
* When we want the order of entries in sorted ascending order.

**Key points:**

1. HashTable is suitable when you are not working in a multithreading environment.

2. HashMap is slightly better than HashTable but it is not thread-safe. It is suitable if the order of elements is not an issue.

3. TreeMap is slower than HashMap but it is suitable when you need the map in sorted ascending order.

4. LinkedHashMap is also slower than HashMap, but it is preferred if more number of insertions and deletions happen on the map.