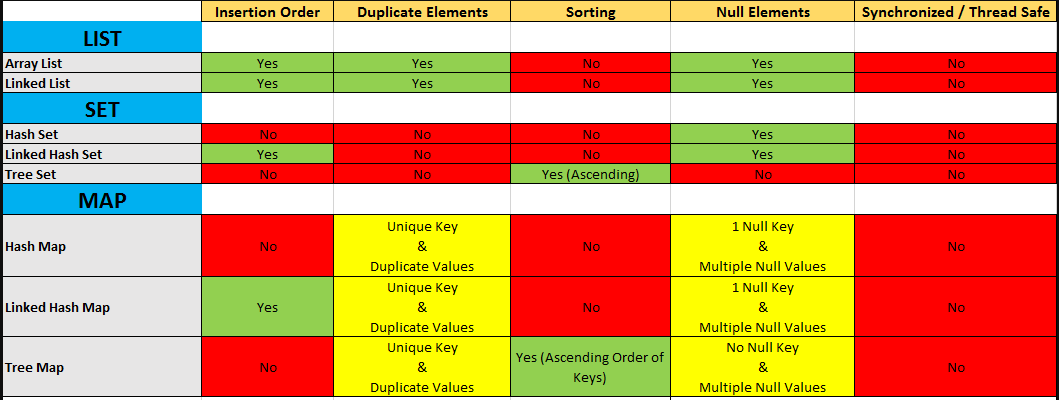
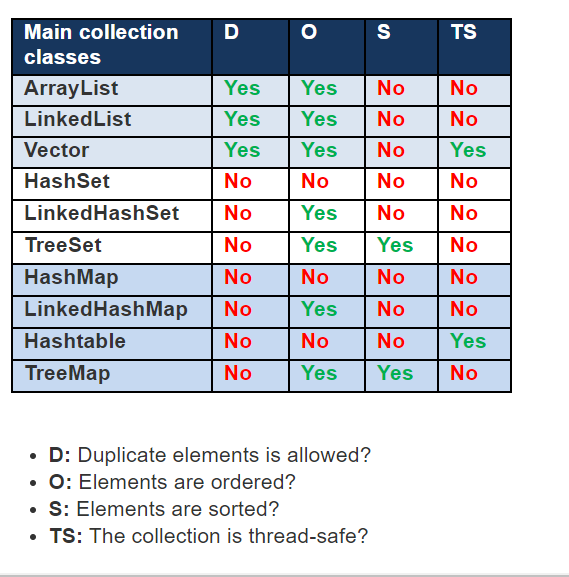
**COLLECTION**





**Collections**

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

**What is Collection in Java**

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

What is a framework in Java

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

**What is Collection framework**

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

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

## **List Interface**

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

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

To instantiate the List interface, we must use :

List <data-type> list1= **new** ArrayList();

List <data-type> list2 = **new** LinkedList();

List <data-type> list3 = **new** Vector();

List <data-type> list4 = **new** Stack();

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

The classes that implement the List interface are given below.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a  **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList  because it uses a doubly linked list, so no bit shifting is  required in memory. |
|  |  |
| 3) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |
| 4) The memory location for the elements of an ArrayList is contiguous. | The location for the elements of a linked list is not contagious. |
| 5) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList. | There is no case of default capacity in a LinkedList.  In LinkedList, an empty list is created when a  LinkedList is initialized. |
|  |  |

## **ArrayList**

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

**import** java.util.\*;

**class** TestJavaCollection1{

**public** **static** **void** main(String args[]){

ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist

list.add("Ravi");//Adding object in arraylist

list.add("Vijay");

list.add("Ravi");

list.add("Ajay");

//Traversing list through Iterator

Iterator itr=list.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Ravi

Vijay

Ravi

Ajay

### **Ways to iterate the elements of the collection in Java**

There are various ways to traverse the collection elements:

1. By Iterator interface.
2. By for-each loop.
3. By ListIterator interface.
4. By for loop.
5. By forEach() method.
6. By forEachRemaining() method.

### **Iterating Collection through remaining ways**

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

**import** java.util.\*;

**class** ArrayList4{

**public** **static** **void** main(String args[]){

    ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist

           list.add("Ravi");//Adding object in arraylist

           list.add("Vijay");

           list.add("Ravi");

           list.add("Ajay");

           System.out.println("Traversing list through List Iterator:");

//Here, element iterates in reverse order

              ListIterator<String> list1=list.listIterator(list.size());

**while**(list1.hasPrevious())

              {

                  String str=list1.previous();

                  System.out.println(str);

              }

        System.out.println("Traversing list through for loop:");

**for**(**int** i=0;i<list.size();i++)

           {

            System.out.println(list.get(i));

           }

**//basic iterator way**

Iterator<String> i= list.iterator (); while(i.hasNext()) { System.out.println(i.next()); }

        System.out.println("Traversing list through forEach() method:");

        //The forEach() method is a new feature, introduced in Java 8.

            list.forEach(a->{ //Here, we are using lambda expression

                System.out.println(a);

              });

            System.out.println("Traversing list through forEachRemaining() method:");

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

              itr.forEachRemaining(a - > //Here, we are using lambda expression

              {

            System.out.println(a);

              });

 }

}

**Output:**

Traversing list through List Iterator:

Ajay

Ravi

Vijay

Ravi

Traversing list through for loop:

Ravi

Vijay

Ravi

Ajay

Traversing list through forEach() method:

Ravi

Vijay

Ravi

Ajay

Traversing list through forEachRemaining() method:

Ravi

Vijay

Ravi

Ajay

**Speed Test Between ArrayList Vs LinkedList**

**import** java.io.IOException;

**import** java.util.ArrayList;

**import** java.util.Iterator;

**import** java.util.LinkedList;

**public** **class** Encapsulation{

**public** **static** **void** main(String[] args){

**long** startTime = System.*currentTimeMillis*();

ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist

**for** (**int** i=0; i<100000; i++){

list.add("Ravi");

list.add("jayanth");

list.remove(0);

}

System.***out***.println("Time taken by ArrayList: " +

(System.*currentTimeMillis*() - startTime) + "ms");

//------------------------------------------------------------

startTime = System.*currentTimeMillis*();

LinkedList<String> ll=**new** LinkedList<String>();

**for** (**int** i=0; i<100000; i++){

ll.add("Ravi");

ll.add("jayanth");

ll.remove(0);

}

System.***out***.println("Time taken by LinkedList: " +

(System.*currentTimeMillis*() - startTime) + "ms");

}

}

## **LinkedList**

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

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection2{

**public** **static** **void** main(String args[]){

LinkedList<String> al=**new** LinkedList<String>();

al.add("Ravi");

al.add("Vijay");

al.add("Ravi");

al.add("Ajay");

Iterator<String> itr=al.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Ravi

Vijay

Ravi

Ajay

### **Java LinkedList example to add elements**

Here, we see different ways to add elements.

**import** java.util.\*;

**public** **class** LinkedList2{

**public** **static** **void** main(String args[]){

 LinkedList<String> ll=**new** LinkedList<String>();

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

           ll.add("Ravi");

           ll.add("Vijay");

           ll.add("Ajay");

           System.out.println("After invoking add(E e) method: "+ll);

           //Adding an element at the specific position

           ll.add(1, "Gaurav");

           System.out.println("After invoking add(int index, E element) method: "+ll);

           LinkedList<String> ll2=**new** LinkedList<String>();

           ll2.add("Sonoo");

           ll2.add("Hanumat");

           //Adding second list elements to the first list

           ll.addAll(ll2);

           System.out.println("After invoking addAll(Collection<? extends E> c) method: "+ll);

           LinkedList<String> ll3=**new** LinkedList<String>();

           ll3.add("John");

           ll3.add("Rahul");

           //Adding second list elements to the first list at specific position

           ll.addAll(1, ll3);

           System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+ll);

           //Adding an element at the first position

           ll.addFirst("Lokesh");

           System.out.println("After invoking addFirst(E e) method: "+ll);

           //Adding an element at the last position

           ll.addLast("Harsh");

           System.out.println("After invoking addLast(E e) method: "+ll);

 }

}

Initial list of elements: []

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

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

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

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

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

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

After invoking addFirst(E e) method:

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

After invoking addLast(E e) method:

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

### **Java LinkedList example to remove elements**

Here, we see different ways to remove an element.

**import** java.util.\*;

**public** **class** LinkedList3 {

**public** **static** **void** main(String [] args)

        {

           LinkedList<String> ll=**new** LinkedList<String>();

           ll.add("Ravi");

           ll.add("Vijay");

           ll.add("Ajay");

           ll.add("Anuj");

           ll.add("Gaurav");

           ll.add("Harsh");

           ll.add("Virat");

           ll.add("Gaurav");

           ll.add("Harsh");

           ll.add("Amit");

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

         //Removing specific element from arraylist

              ll.remove("Vijay");

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

         //Removing element on the basis of specific position

              ll.remove(0);

              System.out.println("After invoking remove(index) method: "+ll);

              LinkedList<String> ll2=**new** LinkedList<String>();

              ll2.add("Ravi");

              ll2.add("Hanumat");

         // Adding new elements to arraylist

              ll.addAll(ll2);

              System.out.println("Updated list : "+ll);

         //Removing all the new elements from arraylist

              ll.removeAll(ll2);

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

         //Removing first element from the list

              ll.removeFirst();

              System.out.println("After invoking removeFirst() method: "+ll);

          //Removing first element from the list

              ll.removeLast();

              System.out.println("After invoking removeLast() method: "+ll);

          //Removing first occurrence of element from the list

              ll.removeFirstOccurrence("Gaurav");

              System.out.println("After invoking removeFirstOccurrence() method: "+ll);

          //Removing last occurrence of element from the list

              ll.removeLastOccurrence("Harsh");

              System.out.println("After invoking removeLastOccurrence() method: "+ll);

              //Removing all the elements available in the list

              ll.clear();

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

       }

    }

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

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

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

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

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

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

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

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

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

After invoking clear() method: []

### **Java LinkedList Example to reverse a list of elements**

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

Output: Ajay

Vijay

Ravi

## **Vector**

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

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection3{

**public** **static** **void** main(String args[]){

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

v.add("Ayush");

v.add("Amit");

v.add("Ashish");

v.add("Garima");

Iterator<String> itr=v.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Ayush

Amit

Ashish

Garima

## **Stack**

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

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection4{

**public** **static** **void** main(String args[]){

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

stack.push("Ayush");

stack.push("Garvit");

stack.push("Amit");

stack.push("Ashish");

stack.push("Garima");

stack.pop();

Iterator<String> itr=stack.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Ayush

Garvit

Amit

Ashish

### **Java ArrayList example to add elements**

Here, we see different ways to add an element.

**import** java.util.\*;

**class** ArrayList7{

**public** **static** **void** main(String args[]){

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

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

           //Adding elements to the end of the list

           al.add("Ravi");

           al.add("Vijay");

           al.add("Ajay");

           System.out.println("After invoking add(E e) method: "+al);

           //Adding an element at the specific position

           al.add(1, "Gaurav");

           System.out.println("After invoking add(int index, E element) method: "+al);

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

           al2.add("Sonoo");

           al2.add("Hanumat");

           //Adding second list elements to the first list

           al.addAll(al2);

           System.out.println("After invoking addAll(Collection<? extends E> c) method: "+al);

           ArrayList<String> al3=**new** ArrayList<String>();

           al3.add("John");

           al3.add("Rahul");

           //Adding second list elements to the first list at specific position

           al.addAll(1, al3);

           System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+al);

 }

}

**Output:**

Initial list of elements: []

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

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

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

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

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

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

### **Java ArrayList example to remove elements**

Here, we see different ways to remove an element.

**import** java.util.\*;

**class** ArrayList8 {

**public** **static** **void** main(String [] args)

        {

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

          al.add("Ravi");

          al.add("Vijay");

          al.add("Ajay");

          al.add("Anuj");

          al.add("Gaurav");

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

          //Removing specific element from arraylist

          al.remove("Vijay");

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

          //Removing element on the basis of specific position

          al.remove(0);

          System.out.println("After invoking remove(index) method: "+al);

          //Creating another arraylist

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

          al2.add("Ravi");

          al2.add("Hanumat");

          //Adding new elements to arraylist

          al.addAll(al2);

          System.out.println("Updated list : "+al);

          //Removing all the new elements from arraylist

          al.removeAll(al2);

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

          //Removing elements on the basis of specified condition

          al.removeIf(str -> str.contains("Ajay"));   //Here, we are using Lambda expression

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

          //Removing all the elements available in the list

          al.clear();

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

       }

    }

**Output:**

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

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

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

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

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

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

After invoking clear() method: []

### **Java ArrayList example of retainAll() method**

**import** java.util.\*;

**class** ArrayList9{

**public** **static** **void** main(String args[]){

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

  al.add("Ravi");

  al.add("Vijay");

  al.add("Ajay");

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

  al2.add("Ravi");

  al2.add("Hanumat");

  al.retainAll(al2);

  System.out.println("iterating the elements after retaining the elements of al2");

  Iterator itr=al.iterator();

**while**(itr.hasNext()){

   System.out.println(itr.next());

  }

 }

}

**Output:**

iterating the elements after retaining the elements of al2

Ravi

### **Java ArrayList example of isEmpty() method**

**import** java.util.\*;

**class** ArrayList10{

**public** **static** **void** main(String [] args)

        {

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

          System.out.println("Is ArrayList Empty: "+al.isEmpty());

          al.add("Ravi");

          al.add("Vijay");

          al.add("Ajay");

          System.out.println("After Insertion");

          System.out.println("Is ArrayList Empty: "+al.isEmpty());

       }

    }

**Output:**

Is ArrayList Empty: true

After Insertion

Is ArrayList Empty: false

## **Set Interface**

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

Set can be instantiated as:

Set<data-type> s1 = **new** HashSet<data-type>();

Set<data-type> s2 = **new** LinkedHashSet<data-type>();

Set<data-type> s3 = **new** TreeSet<data-type>();

## **HashSet**

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

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection7{

**public** **static** **void** main(String args[]){

//Creating HashSet and adding elements

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

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

//Traversing elements

Iterator<String> itr=set.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Vijay

Ravi

Ajay

## **LinkedHashSet**

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

Consider the following example.

**import** java.util.\*;

**public** **class** TestJavaCollection8{

**public** **static** **void** main(String args[]){

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

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

Iterator<String> itr=set.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Ravi

Vijay

Ajay

## **SortedSet Interface**

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

The SortedSet can be instantiated as:

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

## **TreeSet**

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

Consider the following example:

**import** java.util.\*;

**public** **class** TestJavaCollection9{

**public** **static** **void** main(String args[]){

//Creating and adding elements

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

set.add("Ravi");

set.add("Vijay");

set.add("Ravi");

set.add("Ajay");

//traversing elements

Iterator<String> itr=set.iterator();

**while**(itr.hasNext()){

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

}

}

}

Output:

Ajay

Ravi

Vijay

**Map**

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

### **Useful methods of Map interface**

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Methods of Map.Entry interface**  |  |  | | --- | --- | | **Method** | **Description** | | K getKey() | It is used to obtain a key. | | V getValue() | It is used to obtain value. | | int hashCode() | It is used to obtain hashCode. | | V setValue(V value) | It is used to replace the value corresponding to this entry with the specified value. | | boolean equals(Object o) | It is used to compare the specified object with the other existing objects. | | static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>> comparingByKey() | It returns a comparator that compare the objects in natural order on key. | | static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super K> cmp) | It returns a comparator that compare the objects by key using the given Comparator. | | static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>> comparingByValue() | It returns a comparator that compare the objects in natural order on value. | | static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super V> cmp) | It returns a comparator that compare the objects by value using the given Comparator. |  **Java HashMap Example** Let's see a simple example of HashMap to store key and value pair.   1. **import** java.util.\*; 2. **public** **class** HashMapExample1{ 3. **public** **static** **void** main(String args[]){ 4. HashMap<Integer,String> map=**new** HashMap<Integer,String>();//Creating HashMap 5. map.put(1,"1@.");  //Put elements in Map 6. map.put(2,"Apple"); 7. map.put(3,"Banana"); 8. map.put(4,"Grapes"); 10. System.out.println("Iterating Hashmap..."); 11. **for**(Map.Entry m : map.entrySet()){ 12. System.out.println(m.getKey()+" "+m.getValue()); 13. } 14. } 15. }   Iterating Hashmap...  1 Mango  2 Apple  3 Banana  4 Grapes  In this example, we are storing Integer as the key and String as the value, so we are using HashMap<Integer,String> as the type. The put() method inserts the elements in the map.  To get the key and value elements, we should call the getKey() and getValue() methods. The Map.Entry interface contains the getKey() and getValue() methods. But, we should call the entrySet() method of Map interface to get the instance of Map.Entry. **No Duplicate Key on HashMap** You cannot store duplicate keys in HashMap. However, if you try to store duplicate key with another value, it will replace the value.  **import** java.util.\*;  **public** **class** HashMapExample2{  **public** **static** **void** main(String args[]){     HashMap<Integer,String> map=**new** HashMap<Integer,String>();//Creating HashMap     map.put(1,"Mango");  //Put elements in Map     map.put(2,"Apple");     map.put(3,"Banana");     map.put(1,"Grapes"); //trying duplicate key       System.out.println("Iterating Hashmap...");  **for**(Map.Entry m : map.entrySet()){      System.out.println(m.getKey()+" "+m.getValue());     }  }   1. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=HashMapExample2)  Iterating Hashmap...  1 Grapes  2 Apple  3 Banana **Java HashMap example to add() elements** Here, we see different ways to insert elements.   1. **import** java.util.\*; 2. **class** HashMap1{ 3. **public** **static** **void** main(String args[]){ 4. HashMap<Integer,String> hm=**new** HashMap<Integer,String>(); 5. System.out.println("Initial list of elements: "+hm); 6. hm.put(100,"Amit"); 7. hm.put(101,"Vijay"); 8. hm.put(102,"Rahul"); 10. System.out.println("After invoking put() method "); 11. **for**(Map.Entry m:hm.entrySet()){ 12. System.out.println(m.getKey()+" "+m.getValue()); 13. } 15. hm.putIfAbsent(103, "Gaurav"); 16. System.out.println("After invoking putIfAbsent() method "); 17. **for**(Map.Entry m:hm.entrySet()){ 18. System.out.println(m.getKey()+" "+m.getValue()); 19. } 20. HashMap<Integer,String> map=**new** HashMap<Integer,String>(); 21. map.put(104,"Ravi"); 22. map.putAll(hm); 23. System.out.println("After invoking putAll() method "); 24. **for**(Map.Entry m:map.entrySet()){ 25. System.out.println(m.getKey()+" "+m.getValue()); 26. } 27. } 28. }   Initial list of elements: {}  After invoking put() method  100 Amit  101 Vijay  102 Rahul  After invoking putIfAbsent() method  100 Amit  101 Vijay  102 Rahul  103 Gaurav  After invoking putAll() method  100 Amit  101 Vijay  102 Rahul  103 Gaurav  104 Ravi **Java HashMap example to remove() elements** Here, we see different ways to remove elements.   1. **import** java.util.\*; 2. **public** **class** HashMap2 { 3. **public** **static** **void** main(String args[]) { 4. HashMap<Integer,String> map=**new** HashMap<Integer,String>(); 5. map.put(100,"Amit"); 6. map.put(101,"Vijay"); 7. map.put(102,"Rahul"); 8. map.put(103, "Gaurav"); 9. System.out.println("Initial list of elements: "+map); 10. //key-based removal 11. map.remove(100); 12. System.out.println("Updated list of elements: "+map); 13. //value-based removal 14. map.remove(101); 15. System.out.println("Updated list of elements: "+map); 16. //key-value pair based removal 17. map.remove(102, "Rahul"); 18. System.out.println("Updated list of elements: "+map); 19. } 20. }   Output:  Initial list of elements: {100=Amit, 101=Vijay, 102=Rahul, 103=Gaurav}  Updated list of elements: {101=Vijay, 102=Rahul, 103=Gaurav}  Updated list of elements: {102=Rahul, 103=Gaurav}  Updated list of elements: {103=Gaurav} **Java HashMap example to replace() elements** Here, we see different ways to replace elements.   1. **import** java.util.\*; 2. **class** HashMap3{ 3. **public** **static** **void** main(String args[]){ 4. HashMap<Integer,String> hm=**new** HashMap<Integer,String>(); 5. hm.put(100,"Amit"); 6. hm.put(101,"Vijay"); 7. hm.put(102,"Rahul"); 8. System.out.println("Initial list of elements:"); 9. **for**(Map.Entry m:hm.entrySet()) 10. { 11. System.out.println(m.getKey()+" "+m.getValue()); 12. } 13. System.out.println("Updated list of elements:"); 14. hm.replace(102, "Gaurav"); 15. **for**(Map.Entry m:hm.entrySet()) 16. { 17. System.out.println(m.getKey()+" "+m.getValue()); 18. } 19. System.out.println("Updated list of elements:"); 20. hm.replace(101, "Vijay", "Ravi"); 21. **for**(Map.Entry m:hm.entrySet()) 22. { 23. System.out.println(m.getKey()+" "+m.getValue()); 24. } 25. System.out.println("Updated list of elements:"); 26. hm.replaceAll((k,v) -> "Ajay"); 27. **for**(Map.Entry m:hm.entrySet()) 28. { 29. System.out.println(m.getKey()+" "+m.getValue()); 30. } 31. } 32. }   Initial list of elements:  100 Amit  101 Vijay  102 Rahul  Updated list of elements:  100 Amit  101 Vijay  102 Gaurav  Updated list of elements:  100 Amit  101 Ravi  102 Gaurav  Updated list of elements:  100 Ajay  101 Ajay  102 Ajay |