Collections in Java

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

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

Java Collection means a single unit of objects.i.e., a group.

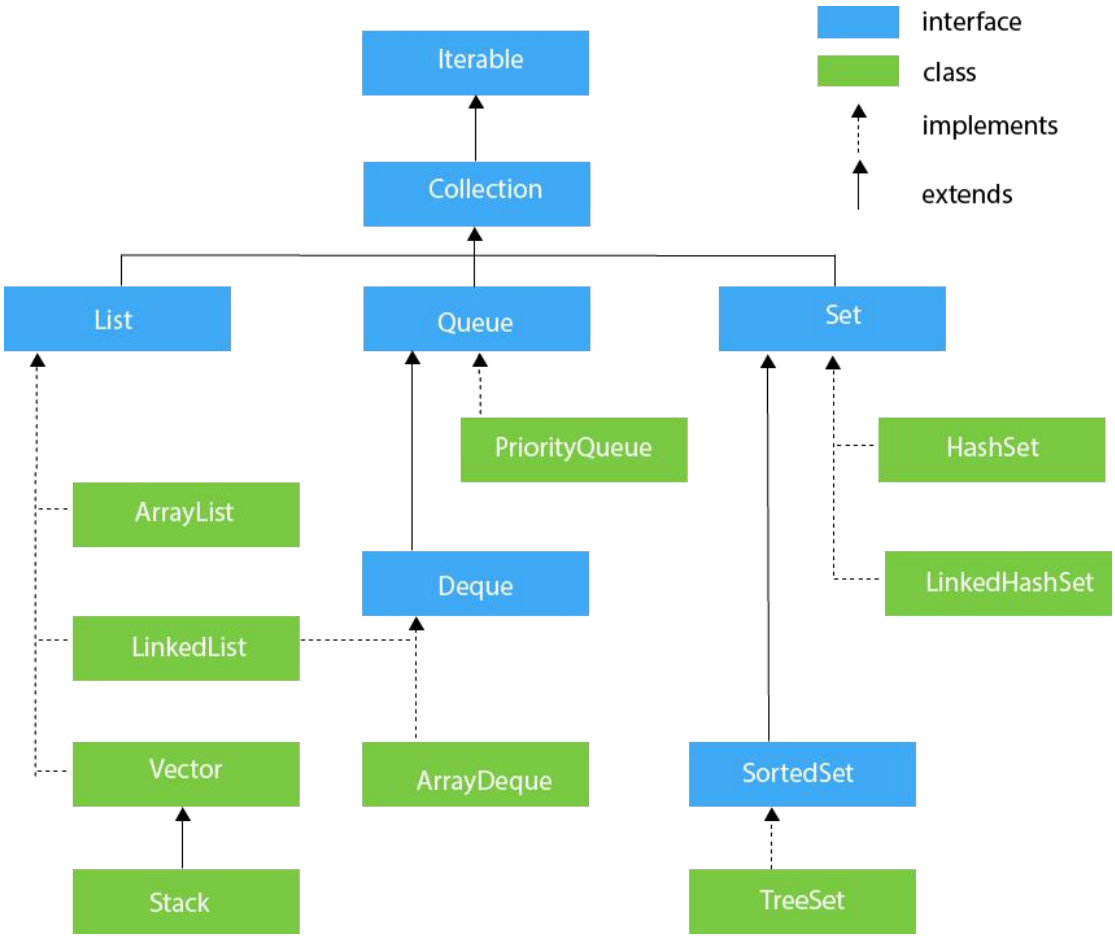
What is a framework in Java

- 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



Iterator interface

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

Methods of Iterator interface

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

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.

3	public void remove()	It removes the last elements returned by the iterator. It is less used.

Iterable Interface

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

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

Collection Interface

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

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

Java Collections class

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

The important points about Java Collections class are:

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

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 :

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

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

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

It inherits the AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

Methods of ArrayList

Method	Description
void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.

boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
void ensureCapacity(int requiredCapacity)	It is used to enhance the capacity of an ArrayList instance.
E get(int index)	It is used to fetch the element from the particular position of the list.
boolean isEmpty()	It returns true if the list is empty, otherwise false.
Iterator()	
listIterator()	
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
Object clone()	It is used to return a shallow copy of an ArrayList.

boolean contains(Object o)	It returns true if the list contains the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
boolean removeIf(Predicate<? super E> filter)	It is used to remove all the elements from the list that satisfies the given predicate.
protected void removeRange(int fromIndex, int toIndex)	It is used to remove all the elements lies within the given range.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.

void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.
Spliterator<E> spliterator()	It is used to create spliterator over the elements in a list.
List<E> subList(int fromIndex, int toIndex)	It is used to fetch all the elements lies within the given range.
int size()	It is used to return the number of elements present in the list.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Java Non-generic Vs. Generic Collection

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

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

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

```
ArrayList list=new ArrayList();//creating old non-generic arraylist
```

```
ArrayList<String> list=new ArrayList<String>();//creating new generic arraylist
```

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

Java ArrayList Example

```
import java.util.*;
 public class ArrayListExample1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating a
rraylist
     list.add("Mango");//Adding object in arraylist
     list.add("Apple");
     list.add("Banana");
     list.add("Grapes");
     //Printing the arraylist object
     System.out.println(list);
  //Traversing list through Iterator
  Iterator itr=list.iterator();//getting the Iterator
  while(itr.hasNext()){//check if iterator has the elements
   System.out.println(itr.next());//printing the element and mo
ve to next
  }
 }
}
```

Iterating ArrayList using For-each loop

```
import java.util.*;
public class ArrayListExample3{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arr
aylist
  list.add("Mango");//Adding object in arraylist
  list.add("Apple");
  list.add("Banana");
  list.add("Grapes");
  //Traversing list through for-each loop
  for(String fruit:list)
    System.out.println(fruit);
  }
}
```

```
        ArrayList<String> al=new ArrayList<String>();
        al.add("Mango");
        al.add("Apple");
        al.add("Banana");
        al.add("Grapes");
        //accessing the element
        System.out.println("Returning element: "+al.get(1));//it will return the 2
nd element, because index starts from 0
        //changing the element
        al.set(1,"Dates");
        //Traversing list
        for(String fruit:al)
          System.out.println(fruit);
        }
```

How to Sort ArrayList

The java.util package provides a utility class **Collections** which has the static method sort(). Using the **Collections.sort()** method, we can easily sort the ArrayList.

```
        Collections.sort(list1);
          //Traversing list through the for-each loop
         for(String fruit:list1)
           System.out.println(fruit);
```

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.

```
        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));
                }
                  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() met
hod:");
                  Iterator<String> itr=list.iterator();
                  itr.forEachRemaining(a-> //Here, we are using lambda expression

                  {
                 System.out.println(a);
                  });
```

User-defined class objects in Java ArrayList

```
class Student{
 int rollno;
 String name;
 int age;
 Student(int rollno,String name,int age){
  this.rollno=rollno;
  this.name=name;
  this.age=age;
 }
```

```
 //Creating user-defined class objects
  Student s1=new Student(101,"Sonoo",23);
  Student s2=new Student(102,"Ravi",21);
  Student s2=new Student(103,"Hanumat",25);
  //creating arraylist
  ArrayList<Student> al=new ArrayList<Student>();
  al.add(s1);//adding Student class object
  al.add(s2);
  al.add(s3);
  //Getting Iterator
  Iterator itr=al.iterator();
  //traversing elements of ArrayList object
  while(itr.hasNext()){
    Student st=(Student)itr.next();
    System.out.println(st.rollno+" "+st.name+" "+st.age);
  }
```

Java LinkedList class

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

The important points about Java LinkedList are:

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

Doubly Linked List

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

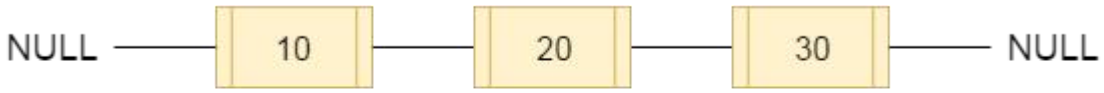


fig- doubly linked list

```
        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());
         }
         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<? exte
nds E> c) method: "+ll);
```

remove elements

```
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);
```

reverse a list of elements

```
//Traversing the list of elements in reverse order
        Iterator i=ll.descendingIterator();
        while(i.hasNext())
        {
           System.out.println(i.next());
        }
```

Difference between ArrayList and LinkedList

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.

3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

Java List

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

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

How to convert Array to List

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

```
        String[] array={"Java","Python","PHP","C++"};
        System.out.println("Printing Array: "+Arrays.toString(array));
        //Converting Array to List
        List<String> list=new ArrayList<String>();
        for(String lang:array){
        list.add(lang);
        }
    System.out.println("Printing List: "+list);
```

convert List to Array

```
        List<String> fruitList = new ArrayList<>();
         fruitList.add("Mango");
         fruitList.add("Banana");
         fruitList.add("Apple");
         fruitList.add("Strawberry");
         //Converting ArrayList to Array
         String[] array = fruitList.toArray(new String[fruitList.size()]);
         System.out.println("Printing Array: "+Arrays.toString(array));
         System.out.println("Printing List: "+fruitList);
).      }
```

Get and Set Element in List

```
        //accessing the element
         System.out.println("Returning element: "+list.get(1));//it will return t
he 2nd element, because index starts from 0
         //changing the element
         list.set(1,"Dates");
         //Iterating the List element using for-each loop
         for(String fruit:list)
          System.out.println(fruit);


         }
```

How to Sort List

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

```
        Collections.sort(list1);
          //Traversing list through the for-each loop
        for(String fruit:list1)
          System.out.println(fruit);
```

Java ListIterator Interface

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

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

```
        List<String> al=new ArrayList<String>();
                al.add("Amit");
                al.add("Vijay");
                al.add("Kumar");
                al.add(1,"Sachin");
                ListIterator<String> itr=al.listIterator();
                System.out.println("Traversing elements in forward direction");
                while(itr.hasNext()){

                System.out.println("index:"+itr.nextIndex()+" value:"+itr.next());
                }
                System.out.println("Traversing elements in backward direction");
                while(itr.hasPrevious()){

                System.out.println("index:"+itr.previousIndex()+" value:"+itr.previou
 s());
                }
        }
```

Java HashSet

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

The important points about Java HashSet class are:

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

Java HashSet Example

```
        //Creating HashSet and adding elements
          HashSet<String> set=new HashSet();
                set.add("One");
                set.add("Two");
                set.add("Three");
               set.add("Three");
                set.add("Three");
                set.add("Four");
                set.add("Five");
                Iterator<String> i=set.iterator();
                while(i.hasNext())
                {
                System.out.println(i.next());
                }
               set.remove("Two");
                System.out.println("After invoking remove(object) method: "+set);
        }
         HashSet<String> set1=new HashSet<String>();
                set1.add("Ajay");
                set1.add("Gaurav");
                set.addAll(set1);
                System.out.println("Updated List: "+set);
                //Removing all the new elements from HashSet
                set.removeAll(set1);
                System.out.println("After invoking removeAll() method: "+set);
                //Removing elements on the basis of specified condition
                set.removeIf(str->str.contains("Vijay"));
                System.out.println("After invoking removeIf() method: "+set);
                //Removing all the elements available in the set
                set.clear();
                System.out.println("After invoking clear() method: "+set);
```

Java LinkedHashSet class

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

The important points about Java LinkedHashSet class are:

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

```
LinkedHashSet<String> al=new LinkedHashSet<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());
 }
```

Java TreeSet class

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

The important points about Java TreeSet class are:

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

```
        TreeSet<String> set=new TreeSet<String>();
              set.add("Ravi");
              set.add("Vijay");
              set.add("Ajay");
              System.out.println("Traversing element through Iterator in descen
ding order");
              Iterator i=set.descendingIterator();
              while(i.hasNext())
              {
                 System.out.println(i.next());
              }
        }
        //Traversing elements
         Iterator<String> itr=al.iterator();
         while(itr.hasNext()){
          System.out.println(itr.next());
         }
```

Java Queue Interface

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

Methods of Java Queue Interface

Method	Description
boolean add(object)	It is used to insert the specified element into this queue and return true upon success.
boolean offer(object)	It is used to insert the specified element into this queue.

Object remove()	It is used to retrieves and removes the head of this queue.
Object poll()	It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.
Object element()	It is used to retrieves, but does not remove, the head of this queue.
Object peek()	It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

PriorityQueue class

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

```
        PriorityQueue<String> queue=new PriorityQueue<String>();
        queue.add("Amit");
        queue.add("Vijay");
        queue.add("Karan");
        queue.add("Jai");
        queue.add("Rahul");
        System.out.println("head:"+queue.element());
        System.out.println("head:"+queue.peek());
        System.out.println("iterating the queue elements:");
        Iterator itr=queue.iterator();
        while(itr.hasNext()){
        System.out.println(itr.next());
        }
        queue.remove();
        queue.poll();
        System.out.println("after removing two elements:");
        Iterator<String> itr2=queue.iterator();
        while(itr2.hasNext()){
        System.out.println(itr2.next());
```

Java Deque Interface

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

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

ArrayDeque class

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

The important points about ArrayDeque class are:

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

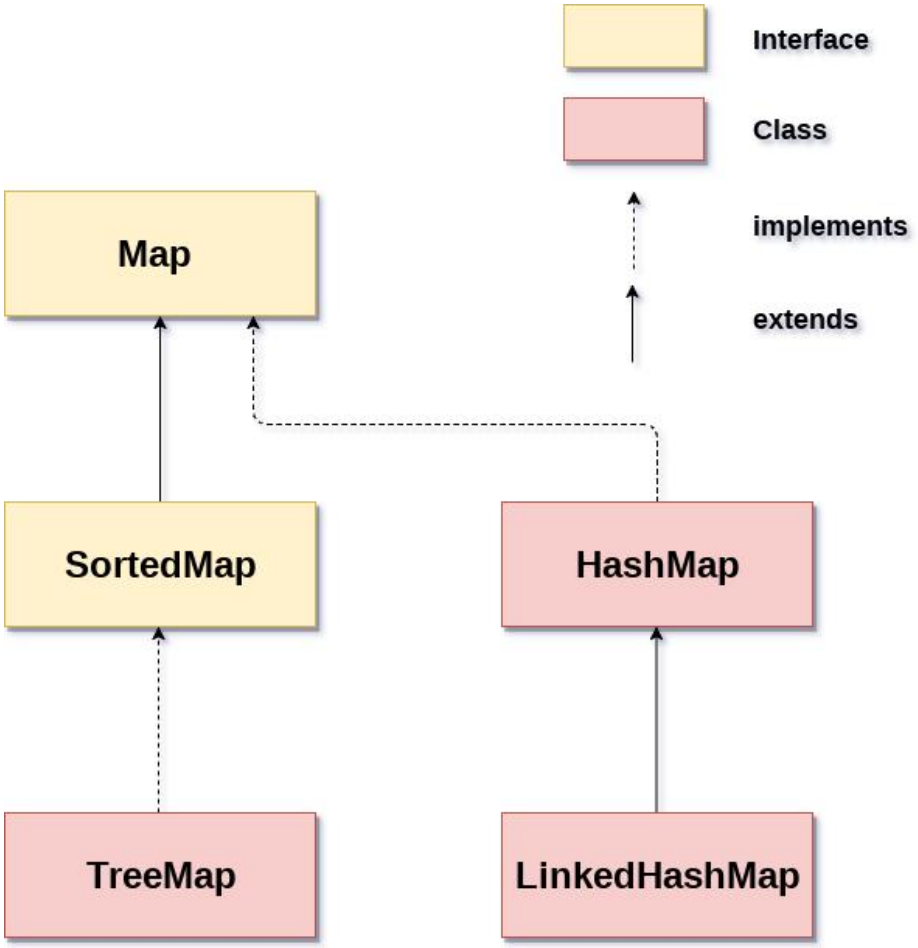
```
        //Creating Deque and adding elements
         Deque<String> deque = new ArrayDeque<String>();
         deque.add("Ravi");
         deque.add("Vijay");
         deque.add("Ajay");
         //Traversing elements
         for (String str : deque) {
         System.out.println(str);
         }
       //deque.poll();
          //deque.pollFirst();//it is same as poll()
          deque.pollLast();
          System.out.println("After pollLast() Traversal...");
          for(String s:deque){
              System.out.println(s);
          }
```

Java Map Interface

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

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

Java Map Hierarchy



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

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

Map.Entry Interface

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

```
Map<Integer,String> map=new HashMap<Integer,String>();
 map.put(100,"Amit");
 map.put(101,"Vijay");
 map.put(102,"Rahul");
 //Elements can traverse in any order
 for(Map.Entry m:map.entrySet()){
  System.out.println(m.getKey()+" "+m.getValue());
 }
```

Java HashMap

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

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

Points to remember

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

```
        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(4,"Grapes");

        System.out.println("Iterating Hashmap...");
        for(Map.Entry m : map.entrySet()){
         System.out.println(m.getKey()+" "+m.getValue());
        }
```

Java HashMap example to add() elements

Here, we see different ways to insert elements.

```
        HashMap<Integer,String> hm=new HashMap<Integer,String>();
          System.out.println("Initial list of elements: "+hm);
           hm.put(100,"Amit");
           hm.put(101,"Vijay");
           hm.put(102,"Rahul");
            System.out.println("After invoking put() method ");
           for(Map.Entry m:hm.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
           }
           hm.putIfAbsent(103, "Gaurav");
           System.out.println("After invoking putIfAbsent() method ");
           for(Map.Entry m:hm.entrySet()){
               System.out.println(m.getKey()+" "+m.getValue());
              }
           HashMap<Integer,String> map=new HashMap<Integer,String>();
           map.put(104,"Ravi");
           map.putAll(hm);
           System.out.println("After invoking putAll() method ");
           for(Map.Entry m:map.entrySet()){
               System.out.println(m.getKey()+" "+m.getValue());
              }
```

remove() elements

```
        //key-based removal
          map.remove(100);
          System.out.println("Updated list of elements: "+map);
          //value-based removal
          map.remove(101);
          System.out.println("Updated list of elements: "+map);
          //key-value pair based removal
          map.remove(102, "Rahul");
          System.out.println("Updated list of elements: "+map);
```

replace() elements

```
        hm.replace(102, "Gaurav");
          for(Map.Entry m:hm.entrySet())
          {
            System.out.println(m.getKey()+" "+m.getValue());
          }
          System.out.println("Updated list of elements:");
          hm.replace(101, "Vijay", "Ravi");
          for(Map.Entry m:hm.entrySet())
          {
            System.out.println(m.getKey()+" "+m.getValue());
          }
```

```
      class Book {
      int id;
      String name,author,publisher;
      int quantity;
      public Book(int id, String name, String author, String publisher, int quantit
y) {
          this.id = id;
          this.name = name;
          this.author = author;
          this.publisher = publisher;
          this.quantity = quantity;
      }
      }
```

```
        public class MapExample {
        public static void main(String[] args) {
            //Creating map of Books
            Map<Integer,Book> map=new HashMap<Integer,Book>();
            //Creating Books
            Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);

            Book b2=new Book(102,"Data Communications & Networking","For
ouzan","Mc Graw Hill",4);
            Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
            //Adding Books to map
            map.put(1,b1);
            map.put(2,b2);
            map.put(3,b3);

            //Traversing map
            for(Map.Entry<Integer, Book> entry:map.entrySet()){
                int key=entry.getKey();
                Book b=entry.getValue();
                System.out.println(key+" Details:");
                System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publish
er+" "+b.quantity);
            }
```

Working of HashMap in Java

What is Hashing

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

What is HashMap

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses

an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.

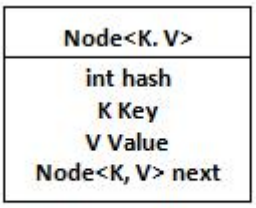


Figure: Representation of a Node

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

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

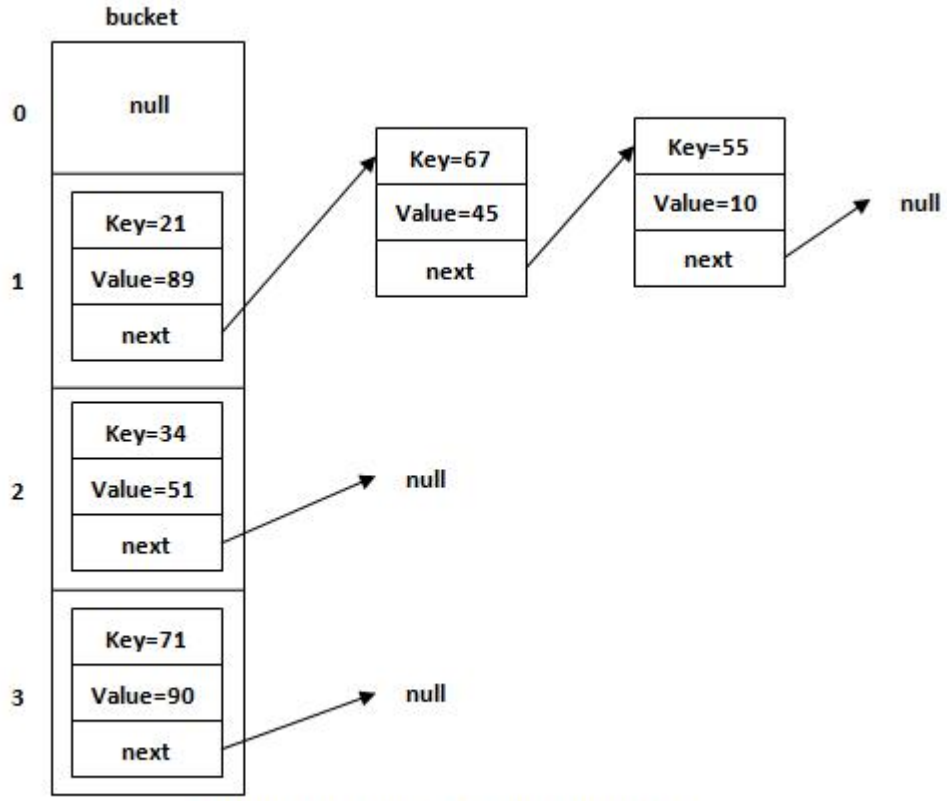


Figure: Allocation of nodes in Bucket

Insert Key, Value pair in HashMap

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

Java LinkedHashMap class

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

Points to remember

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

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

Collection<V> values()	It returns a Collection view of the values contained in this map.

```
        LinkedHashMap<Integer,String> hm=new LinkedHashMap<Integer,S
tring>();

          hm.put(100,"Amit");
          hm.put(101,"Vijay");
          hm.put(102,"Rahul");

        for(Map.Entry m:hm.entrySet()){
          System.out.println(m.getKey()+" "+m.getValue());
          }
```

Java TreeMap class

Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

The important points about Java TreeMap class are:

- Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
- Java TreeMap contains only unique elements.
- Java TreeMap cannot have a null key but can have multiple null values.
- Java TreeMap is non synchronized.
- Java TreeMap maintains ascending order.

TreeMap class declaration

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

```
TreeMap<Integer,String> map=new TreeMap<Integer,String>();

    map.put(100,"Amit");
    map.put(102,"Ravi");
    map.put(101,"Vijay");
    map.put(103,"Rahul");

    for(Map.Entry m:map.entrySet()){
     System.out.println(m.getKey()+" "+m.getValue());
    }
```

What is difference between HashMap and TreeMap?

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

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author, String publisher, int quantity) {

    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}
public class MapExample {
public static void main(String[] args) {
    //Creating map of Books
    Map<Integer,Book> map=new TreeMap<Integer,Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","M
c Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    map.put(2,b2);
    map.put(1,b1);
    map.put(3,b3);
    //Traversing map
    for(Map.Entry<Integer, Book> entry:map.entrySet()){
        int key=entry.getKey();
        Book b=entry.getValue();
        System.out.println(key+" Details:");
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.
quantity);
    }
```

Java Hashtable class

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

Points to remember

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

```
        Hashtable<Integer,String> hm=new Hashtable<Integer,String>();

        hm.put(100,"Amit");
        hm.put(102,"Ravi");
        hm.put(101,"Vijay");
        hm.put(103,"Rahul");

        for(Map.Entry m:hm.entrySet()){
         System.out.println(m.getKey()+" "+m.getValue());
        }
```

Difference between HashMap and Hashtable

HashMap	Hashtable
1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code.	Hashtable is **synchronized**. It is thread-safe and can be shared with many threads.

2) HashMap **allows one null key and multiple null values**.	Hashtable **doesn't allow any null key or value**.
3) HashMap is a **new class introduced in JDK 1.2**.	Hashtable is a **legacy class**.
4) HashMap is **fast**.	Hashtable is **slow**.
5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap);	Hashtable is internally synchronized and can't be unsynchronized.
6) HashMap is **traversed by Iterator**.	Hashtable is **traversed by Enumerator and Iterator**.
7) Iterator in HashMap is **fail-fast**.	Enumerator in Hashtable is **not fail-fast**.
8) HashMap inherits **AbstractMap** class.	Hashtable inherits **Dictionary** class.

Java EnumSet class

Java EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.

```
import java.util.*;
enum days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SA
TURDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);

    // Traversing elements
    Iterator<days> iter = set.iterator();
    while (iter.hasNext())
      System.out.println(iter.next());
  }
}
```

allOf() and noneOf()

```
import java.util.*;
enum days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SAT
URDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set1 = EnumSet.allOf(days.class);
      System.out.println("Week Days:"+set1);
      Set<days> set2 = EnumSet.noneOf(days.class);
      System.out.println("Week Days:"+set2);
  }
}
```

Java Collections Example

```
        List<String> list = new ArrayList<String>();
            list.add("C");
            list.add("Core Java");
            list.add("Advance Java");
            System.out.println("Initial collection value:"+list);
            Collections.addAll(list, "Servlet","JSP");
            System.out.println("After adding elements collection value:"+list
 );
            String[] strArr = {"C#", ".Net"};
            Collections.addAll(list, strArr);
).          System.out.println("After adding array collection value:"+list);
..       }
```

max(),min()

```
        List<Integer> list = new ArrayList<Integer>();
            list.add(46);
            list.add(67);
            list.add(24);
            list.add(16);
            list.add(8);
            list.add(12);
            System.out.println("Value of maximum element from the collecti
on: "+Collections.max(list));
        System.out.println("Value of minimum element from the collection: "
+Collections.min(list));
).          }
```

Java Comparable interface

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named

compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

compareTo(Object obj) method

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

- positive integer, if the current object is greater than the specified object.
- negative integer, if the current object is less than the specified object.
- zero, if the current object is equal to the specified object.

Java Comparable Example

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

```
class Student implements Comparable<Student>{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}
).
..  public int compareTo(Student st){
..  if(age==st.age)
..  return 0;
..  else if(age>st.age)
..  return 1;
..  else
..  return -1;
..  }
..  }
```

```
import java.util.*;
public class TestSort1{
public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

Collections.sort(al);
for(Student st:al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
```

Java Comparator interface

Java Comparator interface is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Java Comparator Example (Generic)

```
class Student{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}
}
```

```
import java.util.*;
class AgeComparator implements Comparator<Student>{
public int compare(Student s1,Student s2){
if(s1.age==s2.age)
return 0;
else if(s1.age>s2.age)
return 1;
else
return -1;
}
```

```
class NameComparator implements Comparator<Student>{
public int compare(Student s1,Student s2){
return s1.name.compareTo(s2.name);
}
}
```

```
 public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
 al.add(new Student(101,"Vijay",23));
 al.add(new Student(106,"Ajay",27));
 al.add(new Student(105,"Jai",21));
   System.out.println("Sorting by Name");
   Collections.sort(al,new NameComparator());
 for(Student st: al){
 System.out.println(st.rollno+" "+st.name+" "+st.age);
 }
   System.out.println("Sorting by age");
   Collections.sort(al,new AgeComparator());
 for(Student st: al){
 System.out.println(st.rollno+" "+st.name+" "+st.age);
 }
```

Properties class in Java

The **properties** object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.

It can be used to get property value based on the property key. The Properties class provides methods to get data from the properties file and store data into the properties file. Moreover, it can be used to get the properties of a system.

An Advantage of the properties file

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

db.properties

```
user=system
password=oracle
```

```
public class Test {
public static void main(String[] args)throws Exception{
   FileReader reader=new FileReader("db.properties");

   Properties p=new Properties();
   p.load(reader);

   System.out.println(p.getProperty("user"));
   System.out.println(p.getProperty("password"));
}
```

Example of Properties class to get all the system properties

```
public class Test {
public static void main(String[] args)throws Exception{

Properties p=System.getProperties();
Set set=p.entrySet();

Iterator itr=set.iterator();
while(itr.hasNext()){
Map.Entry entry=(Map.Entry)itr.next();
System.out.println(entry.getKey()+" = "+entry.getValue());
}
```

Example of Properties class to create the properties file

```
public class Test {
public static void main(String[] args)throws Exception{

Properties p=new Properties();
p.setProperty("name","Sonoo Jaiswal");
p.setProperty("email","sonoojaiswal@javatpoint.com");

p.store(new FileWriter("info.properties"),"Javatpoint Properties Example
");

}
```