

UNIT-4

Collection Framework in Java

Collections in java is a framework that provides an architecture to store and manipulate the group of objects.

All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

What is framework in java

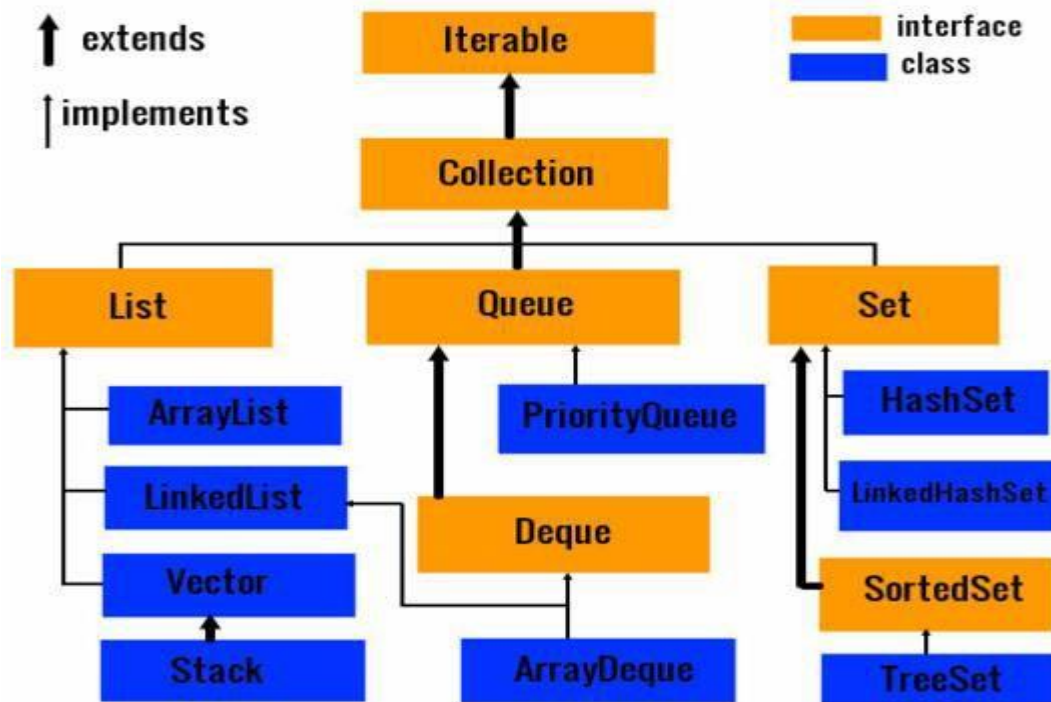
- provides readymade architecture.
- represents set of classes and interface.
- is optional.

What is Collection framework

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

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

Hierarchy of Collection Framework



Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits 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 Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

ArrayList class declaration

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

Constructors of Java ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

Java ArrayList Example

```
import java.util.*;
class TestCollection1 {
    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()); } } }
```

```
Ravi
Vijay
Ravi
Ajay
```

vector

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

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

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

Example of Java Vector

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

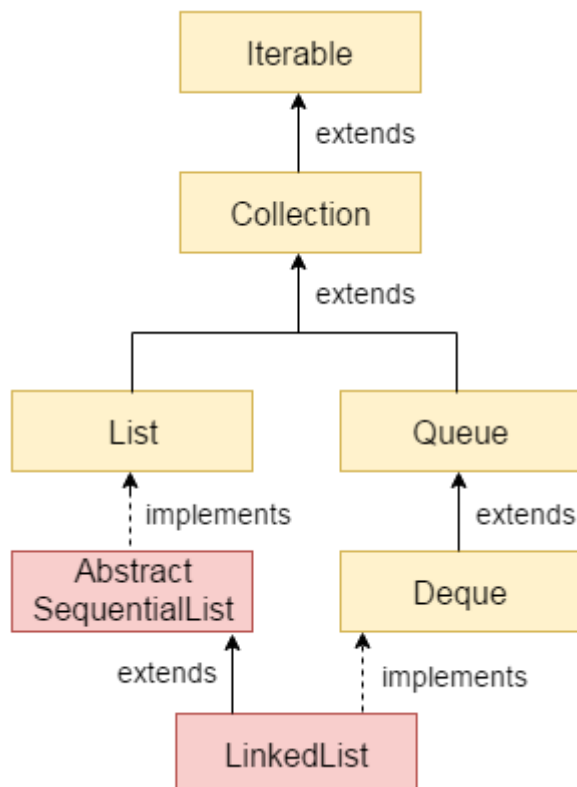
1. **import** java.util.*;
2. **class** TestVector1{
3. **public static void** main(String args[]){
4. Vector<String> v=**new** Vector<String>();//creating vector
5. v.add("umesh");//method of Collection
6. v.addElement("irfan");//method of Vector
7. v.addElement("kumar");
8. //traversing elements using Enumeration

```
9. Enumeration e=v.elements();
10. while(e.hasMoreElements()){
11. System.out.println
(e.nextElement()); 12. }
} }
```

Output:

```
umesh
irfan
kumar
```

Java LinkedList class:



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

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.

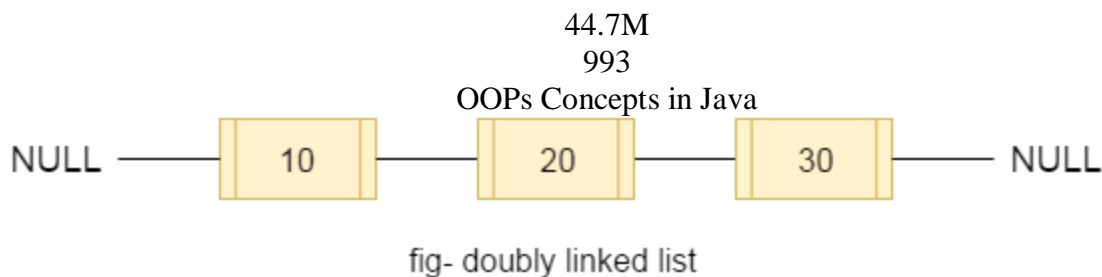
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.

Hierarchy of LinkedList class

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

Doubly Linked List

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



LinkedList class declaration

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

public class LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>, Cloneable, Serializable

Constructors of Java LinkedList

Constructor	Description
LinkedList()	It is used to construct an empty list.
LinkedList(Collection<? extends E> c)	It is used to construct a list containing the elements of the specified collection, in the order, they are returned by the collection's iterator.

Java LinkedList Example:

```
import java.util.*;
public class LinkedList1{
    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 Hashtable class

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

The important points about Java Hashtable class are:

- A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashCode() method. A Hashtable contains values based on the key.
- It contains only unique elements.
- It may have not have any null key or value.
- It is synchronized.

Hashtable class declaration

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

1. **public class** Hashtable<K,V> **extends** Dictionary<K,V> **implements** Map<K,V>, Cloneable, Serializable

Hashtable class Parameters

Let's see the Parameters for java.util.Hashtable class

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

Constructors of Java Hashtable class

Constructor	Description
Hashtable()	It is the default constructor of hash table it instantiates the Hashtable class.
Hashtable(int size)	It is used to accept an integer parameter and creates a hash table that has an initial size specified by integer value size.
Hashtable(int size, float fillRatio)	It is used to create a hash table that has an initial size specified by size and a fill ratio specified by fillRatio.

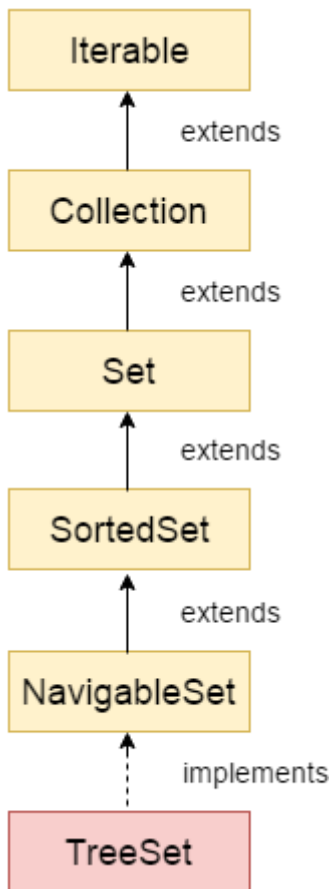
Java Hashtable Example

```
import java.util.*;  
class TestCollection16{  
    public static void main(String args[]){  
        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());  
        } } }
```

Output:

```
103 Rahul  
102 Ravi  
101 Vijay  
100 Amit
```

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 quite fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

Hierarchy of TreeSet class

As shown in the above diagram, Java TreeSet class implements the NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

TreeSet class declaration

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

47.1M

public class TreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>, Cloneable, Serializable

Constructors of Java TreeSet class

Constructor	Description
TreeSet()	It is used to construct an empty tree set that will be sorted in ascending order according to the natural order of the tree set.
TreeSet(Collection<? extends E> c)	It is used to build a new tree set that contains the elements of the collection c.
TreeSet(Comparator<? super E> comparator)	It is used to construct an empty tree set that will be sorted according to given comparator.
TreeSet(SortedSet<E> s)	It is used to build a TreeSet that contains the elements of the given SortedSet.

Java TreeSet Examples

Java TreeSet Example 1:

Let's see a simple example of Java TreeSet.

```
import java.util.*;
class TreeSet1 {
    public static void main(String args[]){
        //Creating and adding elements
        TreeSet<String> al=new TreeSet<String>();
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ravi");
        al.add("Ajay");
        //Traversing elements
        Iterator<String> itr=al.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

Output:

```
Ajay
Ravi
Vijay
```

Java TreeSet Example 2:

Let's see an example of traversing elements in descending order.

```
import java.util.*;
class TreeSet2{
    public static void main(String args[]){
        TreeSet<String> set=new TreeSet<String>();
        set.add("Ravi");
        set.add("Vijay");
        set.add("Ajay");
        System.out.println("Traversing element through Iterator in descending order");
        Iterator i=set.descendingIterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

Output:

```
Traversing element through Iterator in descending order
Vijay
Ravi
Ajay
Traversing element through NavigableSet in descending order
Vijay
Ravi
Ajay
```

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 class declaration

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

public class PriorityQueue<E> **extends** AbstractQueue<E> **implements** Serializable

Java PriorityQueue Example

import java.util.*;

class TestCollection12{

public static void main(String args[]){

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());

}

}

}

Output:head:Amit

head:Amit

iterating the queue elements:

Amit

Jai

Karan

Vijay

Rahul

after removing two elements:

Karan

Rahul

Vijay

ArrayDeque class

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

The important points about ArrayDeque class are:

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

ArrayDeque Hierarchy

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

45.2M
906
C++ vs Java

ArrayDeque class declaration

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

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

Java ArrayDeque Example

```
import java.util.*;
public class ArrayDequeExample {
    public static void main(String[] args) {
        //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);
        }
    }
}
```

Output:

```
Ravi  
Vijay  
Ajay
```

Java ArrayDeque Example: offerFirst() and pollLast()

```
import java.util.*;  
public class DequeExample {  
public static void main(String[] args) {  
    Deque<String> deque=new ArrayDeque<String>();  
    deque.offer("arvind");  
    deque.offer("vimal");  
    deque.add("mukul");  
    deque.offerFirst("jai");  
    System.out.println("After offerFirst Traversal...");  
    for(String s:deque){  
        System.out.println(s);  
    }  
    //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);  
    }  
}  
}
```

Output:

```
After offerFirst Traversal...  
jai  
arvind  
vimal  
mukul  
After pollLast() Traversal...  
jai  
arvind  
vimal
```

Java ArrayDeque Example: Book

```
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 ArrayDequeExample {
public static void main(String[] args) {
    Deque<Book> set=new ArrayDeque<Book>();
    //Creating Books
    Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
    Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
    //Adding Books to Deque
    set.add(b1);
    set.add(b2);
    set.add(b3);
    //Traversing ArrayDeque
    for(Book b:set){
        System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Output:

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```


Iterator

It is a **universal** iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations. It is improved version of Enumeration with additional functionality of remove-ability of a element.

Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface. Iterator is the **only** cursor available for entire collection framework.

Iterator object can be created by calling *iterator()* method present in Collection interface.

```
// Here "c" is any Collection object. itr is of
```

```
// type Iterator interface and refers to "c"
```

```
Iterator itr = c.iterator();
```

Iterator interface defines **three** methods:

```
// Returns true if the iteration has more elements
```

```
public boolean hasNext();
```

```
// Returns the next element in the iteration
```

```
// It throws NoSuchElementException if no more
```

```
// element present
```

```
public Object next();
```

```
// Remove the next element in the iteration
```

```
// This method can be called only once per call
```

```
// to next()
```

```
public void remove();
```

remove() method can throw two exceptions

- *UnsupportedOperationException* : If the remove operation is not supported by this iterator
- *IllegalStateException* : If the next method has not yet been called, or the remove method has already been called after the last call to the next method

Limitations of Iterator:

- Only forward direction iterating is possible.
- Replacement and addition of new element is not supported by Iterator.

How to use Java Iterator?

When a user needs to use the Java Iterator, then it's compulsory for them to make an instance of the Iterator interface from the collection of objects they desire to traverse over. After that, the received Iterator maintains the trail of the components in the underlying collection to make sure that the user will traverse over each of the elements of the collection of objects.

If the user modifies the underlying collection while traversing over an Iterator leading to that collection, then the Iterator will typically acknowledge it and will throw an exception in the next time when the user will attempt to get the next component from the Iterator.

Java Iterator Methods

The following figure perfectly displays the class diagram of the Java Iterator interface. It contains a total of four methods that are:

- hasNext()
- next()
- remove()
- forEachRemaining()

The **forEachRemaining()** method was added in the Java 8. Let's discuss each method in detail.

boolean hasNext(): The method does not accept any parameter. It returns true if there are more elements left in the iteration. If there are no more elements left, then it will return false.

If there are no more elements left in the iteration, then there is no need to call the next() method. In simple words, we can say that the method is used to determine whether the next() method is to be called or not.

E next(): It is similar to hasNext() method. It also does not accept any parameter. It returns E, i.e., the next element in the traversal. If the iteration or collection of objects has no more elements left to iterate, then it throws the NoSuchElementException.

default void remove(): This method also does not require any parameters. There is no return type of this method. The main function of this method is to remove the last element returned by the iterator traversing through the underlying collection. The remove () method can be requested hardly once per the next () method call. If the iterator does not support the remove operation, then it throws the UnsupportedOperationException. It also throws the IllegalStateException if the next method is not yet called.

default void forEachRemaining(Consumer action): It is the only method of Java Iterator that takes a parameter. It accepts action as a parameter. Action is nothing but that is to be performed. There is no return type of the method. This method performs the particularized operation on all of the left components of the collection until all the components are consumed or the action throws an exception. Exceptions thrown by action are delivered to the caller. If the action is null, then it throws a NullPointerException.

Example of Java Iterator

Now it's time to execute a Java program to illustrate the advantage of the Java Iterator interface. The below code produces an ArrayList of city names. Then we initialize an iterator applying the iterator () method of the ArrayList. After that, the list is traversed to represent each element.

JavaIteratorExample.java

```
import java.io.*;
import java.util.*;

public class JavaIteratorExample {
    public static void main(String[] args)
    {
        ArrayList<String> cityNames = new ArrayList<String>();

        cityNames.add("Delhi");
        cityNames.add("Mumbai");
        cityNames.add("Kolkata");
        cityNames.add("Chandigarh");
        cityNames.add("Noida");

        // Iterator to iterate the cityNames
        Iterator iterator = cityNames.iterator();

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

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

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

Output:

```
CityNames elements:
Delhi Mumbai Kolkata Chandigarh Noida
```

Java For-each Loop | Enhanced For Loop

The Java for-each loop or enhanced for loop is introduced since J2SE 5.0. It provides an alternative approach to traverse the array or collection in Java. It is mainly used to traverse the array or collection elements. The advantage of the for-each loop is that it eliminates the possibility of bugs and makes the code more readable. It is known as the for-each loop because it traverses each element one by one.

The drawback of the enhanced for loop is that it cannot traverse the elements in reverse order. Here, you do not have the option to skip any element because it does not work on an index basis. Moreover, you cannot traverse the odd or even elements only.

But, it is recommended to use the Java for-each loop for traversing the elements of array and collection because it makes the code readable.

Advantages

- It makes the code more readable.
- It eliminates the possibility of programming errors.

Syntax

The syntax of Java for-each loop consists of data_type with the variable followed by a colon (:), then array or collection.

```
for(data_type variable : array | collection){  
    //body of for-each loop  
}
```

How it works?

The Java for-each loop traverses the array or collection until the last element. For each element, it stores the element in the variable and executes the body of the for-each loop.

For-each loop Example: Traversing the array elements

//An example of Java for-each loop

```
class ForEachExample1{  
    public static void main(String args[]){  
        //declaring an array  
        int arr[]={ 12,13,14,44};  
        //traversing the array with for-each loop  
        for(int i:arr){  
            System.out.println(i);  
        }  
    }  
}
```

Output:

```
12  
13  
14  
44
```

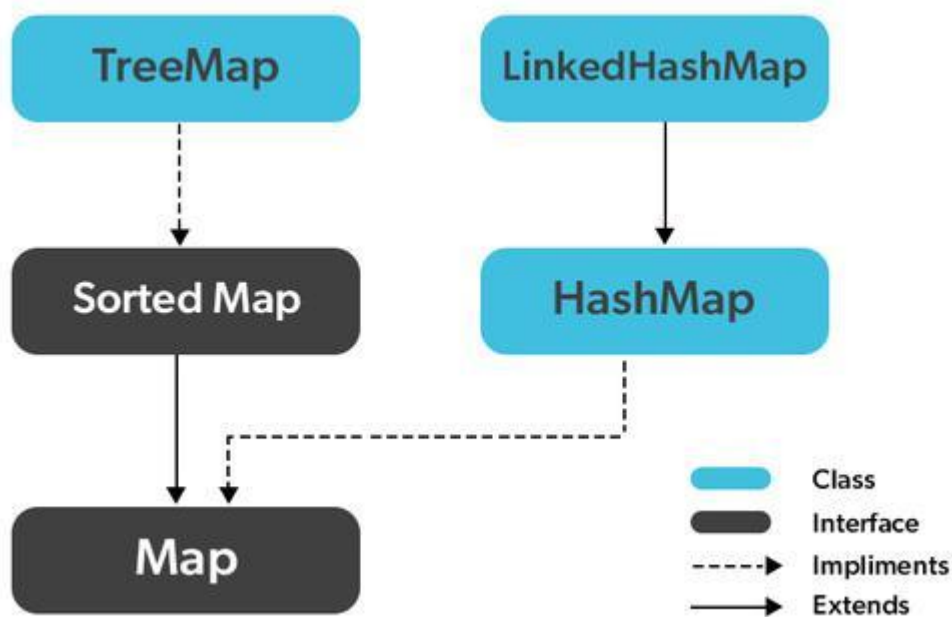
Map Interface in Java

The map interface is present in `java.util` package represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit differently from the rest of the collection types. A map contains unique keys.

Geeks, the brainstormer should have been **why and when to use Maps?**

Maps are perfect to use for key-value association mapping such as dictionaries. The maps are used to perform lookups by keys or when someone wants to retrieve and update elements by keys. Some common scenarios are as follows:

- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).



Creating Map Objects

Since Map is an interface, objects cannot be created of the type map. We always need a class that extends this map in order to create an object. And also, after the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the Map.

Syntax: Defining Type-safe Map

```
Map hm = new HashMap();
```

```
// Obj is the type of the object to be stored in Map
```

Characteristics of a Map Interface

1. A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null values like the HashMap and LinkedHashMap, but some do not like the TreeMap.
2. The order of a map depends on the specific implementations. For example, TreeMap and LinkedHashMap have predictable orders, while HashMap does not.
3. There are two interfaces for implementing Map in java. They are Map and SortedMap, and three classes: HashMap, TreeMap, and LinkedHashMap.

Methods in Map Interface

Method	Action Performed
<u>clear()</u>	This method is used to clear and remove all of the elements or mappings from a specified Map collection.
<u>containsKey(Object)</u>	This method is used to check whether a particular key is being mapped into the Map or not. It takes the key element as a parameter and returns True if that element is mapped in the map.
<u>containsValue(Object)</u>	This method is used to check whether a particular value is being mapped by a single or more than one key in the Map. It takes the value as a parameter and returns True if that value is mapped by any of the key in the map.
<u>entrySet()</u>	This method is used to create a set out of the same elements contained in the map. It basically returns a set view of the map or we can create a new set and store the map elements into them.
<u>equals(Object)</u>	This method is used to check for equality between two maps. It verifies whether the elements of one map passed as a parameter is equal to the elements of this map or not.
<u>get(Object)</u>	This method is used to retrieve or fetch the value mapped by a particular key mentioned in the parameter. It returns NULL when the map contains no such mapping for the key.
<u>hashCode()</u>	This method is used to generate a hashCode for the given map containing keys and values.
<u>isEmpty()</u>	This method is used to check if a map is having any entry for key and value pairs. If no mapping exists, then this returns true.
<u>keySet()</u>	This method is used to return a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.
<u>put(Object, Object)</u>	This method is used to associate the specified value with the specified key in this map.

Method

Action Performed

putAll(Map)

This method is used to copy all of the mappings from the specified map to this map.

remove(Object)

This method is used to remove the mapping for a key from this map if it is present in the map.

size()

This method is used to return the number of key/value pairs available in the map.

values()

This method is used to create a collection out of the values of the map. It basically returns a Collection view of the values in the HashMap.

getOrDefault(Object key, V
defaultValue)

Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.

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.

putIfAbsent(K key, V value)

If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the curassociaterent value.

Example:

- Java

```
// Java Program to Demonstrate
```

```
// Working of Map interface
```

```
// Importing required classes
```

```
import java.util.*;
```

```
// Main class
```

```
class GFG {
```

```
// Main driver method

public static void main(String args[])

{

    // Creating an empty HashMap

    Map<String, Integer> hm

        = new HashMap<String, Integer>();


    // Inserting pairs in above Map

    // using put() method

    hm.put("a", new Integer(100));

    hm.put("b", new Integer(200));

    hm.put("c", new Integer(300));

    hm.put("d", new Integer(400));


    // Traversing through Map using for-each loop

    for (Map.Entry<String, Integer> me :

        hm.entrySet()) {


        // Printing keys

        System.out.print(me.getKey() + ":");

        System.out.println(me.getValue());
```

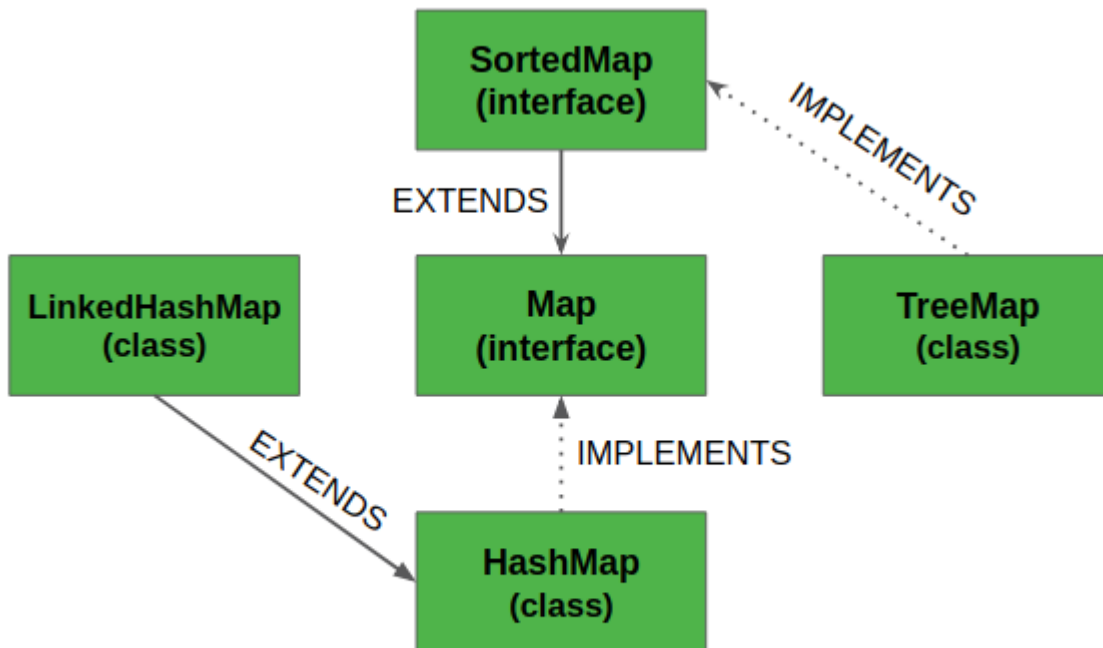


```
}  
  
}  
  
}
```

Output:

a:100
b:200
c:300
d:400

Classes that implement the Map interface are depicted in the below media and described later as follows:



MAP Hierarchy in Java

Class 1: HashMap

HashMap is a part of Java's collection since Java 1.2. It provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value one must know its key. This class uses a technique called Hashing. Hashing is a technique of converting a large String to a small String that represents the same String. A shorter value helps in indexing and faster searches. Let's see how to create a map object using this class.

Example

- Java

```
// Java Program to illustrate the Hashmap Class
```

```
// Importing required classes
```

```
import java.util.*;
```

```
// Main class
```

```
public class GFG {
```

```
    // Main driver method
```

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

```
    {
```

```
        // Creating an empty HashMap
```

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

```
        // Inserting entries in the Map
```

```
        // using put() method
```

```
        map.put("vishal", 10);
```

```
        map.put("sachin", 30);
```

```
        map.put("vaibhav", 20);
```

```

// Iterating over Map

for (Map.Entry<String, Integer> e : map.entrySet())

    // Printing key-value pairs

    System.out.println(e.getKey() + " "

        + e.getValue());

}

}

```

Output

vaibhav 20

vishal 10

sachin 30

Class 2: LinkedHashMap

LinkedHashMap is just like HashMap with an additional feature of maintaining an order of elements inserted into it. HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order. Let's see how to create a map object using this class.

Example

- Java

```
// Java Program to Illustrate the LinkedHashMap Class
```

```
// Importing required classes
```

```
import java.util.*;
```

```
// Main class
```

```
public class GFG {  
  
    // Main driver method  
  
    public static void main(String[] args)  
    {  
  
        // Creating an empty LinkedHashMap  
  
        Map<String, Integer> map = new LinkedHashMap<>();  
  
        // Inserting pair entries in above Map  
  
        // using put() method  
  
        map.put("vishal", 10);  
  
        map.put("sachin", 30);  
  
        map.put("vaibhav", 20);  
  
        // Iterating over Map  
  
        for (Map.Entry<String, Integer> e : map.entrySet())  
  
            // Printing key-value pairs  
  
            System.out.println(e.getKey() + " "  
  
                                + e.getValue());  
  
    }  
}
```

```
}
```

Output:

vishal 10

sachin 30

vaibhav 20

Class 3: TreeMap

The TreeMap in Java is used to implement the Map interface and NavigableMap along with the Abstract Class. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used. This proves to be an efficient way of sorting and storing the key-value pairs. The storing order maintained by the treemap must be consistent with equals just like any other sorted map, irrespective of the explicit comparators. Let's see how to create a map object using this class.

Example

- Java

```
// Java Program to Illustrate TreeMap Class
```

```
// Importing required classes
```

```
import java.util.*;
```

```
// Main class
```

```
public class GFG {
```

```
    // Main driver method
```

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

```
    {
```

```
        // Creating an empty TreeMap
```

```

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

// Inserting custom elements in the Map

// using put() method

map.put("vishal", 10);

map.put("sachin", 30);

map.put("vaibhav", 20);


// Iterating over Map using for each loop

for (Map.Entry<String, Integer> e : map.entrySet())

    // Printing key-value pairs

    System.out.println(e.getKey() + " "

        + e.getValue());

}

}

```

Output:

```

sachin 30
vaibhav 20
vishal 10

```

Comparator Interface in Java with Examples :

A comparator interface is used to order the objects of user-defined classes. A comparator object is capable of comparing two objects of the same class. Following function compare obj1 with obj2.

Syntax:

```
public int compare(Object obj1, Object obj2):
```

Suppose we have an Array/ArrayList of our own class type, containing fields like roll no, name, address, DOB, etc, and we need to sort the array based on Roll no or name?

Method 1: One obvious approach is to write our own sort() function using one of the standard algorithms. This solution requires rewriting the whole sorting code for different criteria like Roll No. and Name.

Method 2: Using comparator interface- Comparator interface is used to order the objects of a user-defined class. This interface is present in java.util package and contains 2 methods compare(Object obj1, Object obj2) and equals(Object element). Using a comparator, we can sort the elements based on data members. For instance, it may be on roll no, name, age, or anything else.

Method of Collections class for sorting List elements is used to sort the elements of List by the given comparator.

```
public void sort(List list, ComparatorClass c)
```

To sort a given List, ComparatorClass must implement a Comparator interface.

How do the sort() method of Collections class work?

Internally the Sort method does call Compare method of the classes it is sorting. To compare two elements, it asks “Which is greater?” Compare method returns -1, 0, or 1 to say if it is less than, equal, or greater to the other. It uses this result to then determine if they should be swapped for their sort.

Example

- Java

```
// Java Program to Demonstrate Working of
```

```
// Comparator Interface
```

```
// Importing required classes
```

```
import java.io.*;
```

```
import java.lang.*;
```

```
import java.util.*;
```

```
// Class 1
```

```
// A class to represent a Student
```

```
class Student {
```

```
// Attributes of a student
```

```
int rollno;
```

```
String name, address;
```

```
// Constructor
```

```
public Student(int rollno, String name, String address)
```

```
{
```

```
    // This keyword refers to current instance itself
```

```
    this.rollno = rollno;
```

```
    this.name = name;
```

```
    this.address = address;
```

```
}
```

```
// Method of Student class
```

```
// To print student details in main()
```

```
public String toString()
```

```
{
```

```
    // Returning attributes of Student
```

```
    return this.rollno + " " + this.name + " "
```



```
        + this.address;

    }

}

// Class 2

// Helper class implementing Comparator interface

class Sortbyroll implements Comparator<Student> {

    // Method

    // Sorting in ascending order of roll number

    public int compare(Student a, Student b)

    {

        return a.rollno - b.rollno;

    }

}
```

```
// Class 3

// Helper class implementing Comparator interface

class Sortbyname implements Comparator<Student> {

    // Method
```

```

// Sorting in ascending order of name

public int compare(Student a, Student b)

{

    return a.name.compareTo(b.name);

}

}

// Class 4

// Main class

class GFG {

    // Main driver method

    public static void main(String[] args)

    {

        // Creating an empty ArrayList of Student type

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

        // Adding entries in above List

        // using add() method

        ar.add(new Student(111, "Mayank", "london"));

```

```
ar.add(new Student(131, "Anshul", "nyc"));

ar.add(new Student(121, "Solanki", "jaipur"));

ar.add(new Student(101, "Aggarwal", "Hongkong"));
```

```
// Display message on console for better readability
```

```
System.out.println("Unsorted");
```

```
// Iterating over entries to print them
```

```
for (int i = 0; i < ar.size(); i++)
```

```
    System.out.println(ar.get(i));
```

```
// Sorting student entries by roll number
```

```
Collections.sort(ar, new Sortbyroll());
```

```
// Display message on console for better readability
```

```
System.out.println("\nSorted by rollno");
```

```
// Again iterating over entries to print them
```

```
for (int i = 0; i < ar.size(); i++)
```

```
    System.out.println(ar.get(i));
```

```
// Sorting student entries by name
```

```
Collections.sort(ar, new Sortbyname());

// Display message on console for better readability

System.out.println("\nSorted by name");


// // Again iterating over entries to print them

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

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

}

}
```

Output

Unsorted

111 Mayank london
131 Anshul nyc
121 Solanki jaipur
101 Aggarwal Hongkong

Sorted by rollno

101 Aggarwal Hongkong
111 Mayank london
121 Solanki jaipur
131 Anshul nyc

Sorted by name

101 Aggarwal Hongkong
131 Anshul nyc
111 Mayank london
121 Solanki jaipur

By changing the return value inside the compare method, you can sort in any order that you wish to, for example: For descending order just change the positions of 'a' and 'b' in the above compare method.

Sort collection by more than one field

In the previous example, we have discussed how to sort the list of objects on the basis of a single field using Comparable and Comparator interface. But, what if we have a requirement to sort ArrayList objects in accordance with more than one field like firstly, sort according to the student name and secondly, sort according to student age.

Example

- Java

```
// Java Program to Demonstrate Working of
```

```
// Comparator Interface Via More than One Field
```

```
// Importing required classes
```

```
import java.util.ArrayList;
```

```
import java.util.Collections;
```

```
import java.util.Comparator;
```

```
import java.util.Iterator;
```

```
import java.util.List;
```

```
// Class 1
```

```
// Helper class representing a Student
```

```
class Student {
```

```
    // Attributes of student
```

```
    String Name;
```

```
    int Age;
```

```
// Parameterized constructor
```

```
public Student(String Name, Integer Age)
```

```
{
```

```
    // This keyword refers to current instance itself
```

```
    this.Name = Name;
```

```
    this.Age = Age;
```

```
}
```

```
// Getter setter methods
```

```
public String getName() { return Name; }
```

```
public void setName(String Name) { this.Name = Name; }
```

```
public Integer getAge() { return Age; }
```

```
public void setAge(Integer Age) { this.Age = Age; }
```

```
// Method
```

```
// Overriding toString() method
```

```
@Override public String toString()
```

```
{  
  
    return "Customer{"  
  
        + "Name=" + Name + ", Age=" + Age + '}'  
  
}
```

// Class 2

// Helper class implementing Comparator interface

static class CustomerSortingComparator

implements Comparator<Student> {

// Method 1

// To compare customers

@Override

public int compare(Student customer1,

Student customer2)

{

// Comparing customers

int NameCompare = customer1.getName().compareTo(

customer2.getName());

int AgeCompare = customer1.getAge().compareTo(

```
customer2.getAge());
```

```
// 2nd level comparison
```

```
return (NameCompare == 0) ? AgeCompare
```

```
    : NameCompare;
```

```
}
```

```
}
```

```
// Method 2
```

```
// Main driver method
```

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

```
{
```

```
    // Create an empty ArrayList
```

```
    // to store Student
```

```
    List<Student> al = new ArrayList<>();
```

```
    // Create customer objects
```

```
    // using constructor initialization
```

```
    Student obj1 = new Student("Ajay", 27);
```

```
    Student obj2 = new Student("Sneha", 23);
```

```
    Student obj3 = new Student("Simran", 37);
```



```
Student obj4 = new Student("Ajay", 22);
```

```
Student obj5 = new Student("Ajay", 29);
```

```
Student obj6 = new Student("Sneha", 22);
```

```
// Adding customer objects to ArrayList
```

```
// using add() method
```

```
al.add(obj1);
```

```
al.add(obj2);
```

```
al.add(obj3);
```

```
al.add(obj4);
```

```
al.add(obj5);
```

```
al.add(obj6);
```

```
// Iterating using Iterator
```

```
// before Sorting ArrayList
```

```
Iterator<Student> custIterator = al.iterator();
```

```
// Display message
```

```
System.out.println("Before Sorting:\n");
```

```
// Holds true till there is single element
```

```
// remaining in List
```

```

while (custIterator.hasNext()) {

    // Iterating using next() method

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

}

// Sorting using sort method of Collections class

Collections.sort(al,

    new CustomerSortingComparator());

// Display message only

System.out.println("\n\nAfter Sorting:\n");

// Iterating using enhanced for-loop

// after Sorting ArrayList

for (Student customer : al) {

    System.out.println(customer);

}

}

}

```

Output

Before Sorting:

Customer{Name=Ajay, Age=27}

```
Customer{Name=Sneha, Age=23}
Customer{Name=Simran, Age=37}
Customer{Name=Ajay, Age=22}
Customer{Name=Ajay, Age=29}
Customer{Name=Sneha, Age=22}
```

After Sorting:

```
Customer{Name=Ajay, Age=22}
Customer{Name=Ajay, Age=27}
Customer{Name=Ajay, Age=29}
Customer{Name=Simran, Age=37}
Customer{Name=Sneha, Age=22}
Customer{Name=Sneha, Age=23}
```

Collection algorithms in java

The java collection framework defines several algorithms as static methods that can be used with collections and map objects.

All the collection algorithms in the java are defined in a class called **Collections** which defined in the **java.util** package.

All these algorithms are highly efficient and make coding very easy. It is better to use them than trying to re-implement them.

The collection framework has the following methods as algorithms.

Method	Description
void sort(List list)	Sorts the elements of the list as determined by their natural ordering.
void sort(List list, Comparator comp)	Sorts the elements of the list as determined by Comparator comp.

Method	Description
void reverse(List list)	Reverses all the elements sequence in list.
void rotate(List list, int n)	Rotates list by n places to the right. To rotate left, use a negative value for n.
void shuffle(List list)	Shuffles the elements in list.
void shuffle(List list, Random r)	Shuffles the elements in the list by using r as a source of random numbers.
void copy(List list1, List list2)	Copies the elements of list2 to list1.
List nCopies(int num, Object obj)	Returns num copies of obj contained in an immutable list. num can not be zero or negative.
void swap(List list, int idx1, int idx2)	Exchanges the elements in the list at the indices specified by idx1 and idx2.
int binarySearch(List list, Object value)	Returns the position of value in the list (must be in the sorted order), or -1 if value is not found.
int binarySearch(List list, Object value, Comparator c)	Returns the position of value in the list ordered according to c, or -1 if value is not found.
int indexOfSubList(List list, List subList)	Returns the index of the first match of subList in the list, or -1 if no match is found.
int lastIndexOfSubList(List list, List subList)	Returns the index of the last match of subList in the list, or -1 if no match is found.
Object max(Collection c)	Returns the largest element from the collection c as determined by natural ordering.
Object max(Collection c, Comparator comp)	Returns the largest element from the collection c as determined by Comparator comp.

Method	Description
Object min(Collection c)	Returns the smallest element from the collection c as determined by natural ordering.
Object min(Collection c, Comparator comp)	Returns the smallest element from the collection c as determined by Comparator comp.
void fill(List list, Object obj)	Assigns obj to each element of the list.
boolean replaceAll(List list, Object old, Object new)	Replaces all occurrences of old with new in the list.
Enumeration enumeration(Collection c)	Returns an enumeration over Collection c.
ArrayList list(Enumeration enum)	Returns an ArrayList that contains the elements of enum.
Set singleton(Object obj)	Returns obj as an immutable set.
List singletonList(Object obj)	Returns obj as an immutable list.
Map singletonMap(Object k, Object v)	Returns the key(k)/value(v) pair as an immutable map.
Collection synchronizedCollection(Collection c)	Returns a thread-safe collection backed by c.
List synchronizedList(List list)	Returns a thread-safe list backed by list.
Map synchronizedMap(Map m)	Returns a thread-safe map backed by m.
SortedMap synchronizedSortedMap(SortedMap sm)	Returns a thread-safe SortedMap backed by sm.
Set synchronizedSet(Set s)	Returns a thread-safe set backed by s.
SortedSet synchronizedSortedSet(SortedSet ss)	Returns a thread-safe set backed by ss.
<div> Department of CSE Page 45 of 70 </div>	

Method	Description
Collection unmodifiableCollection(Collection c)	Returns an unmodifiable collection backed by c.
List unmodifiableList(List list)	Returns an unmodifiable list backed by list.
Set unmodifiableSet(Set s)	Returns an unmodifiable thread-safe set backed by s.
SortedSet unmodifiableSortedSet(SortedSet ss)	Returns an unmodifiable set backed by ss.
Map unmodifiableMap(Map m)	Returns an unmodifiable map backed by m.
SortedMap unmodifiableSortedMap(SortedMap sm)	Returns an unmodifiable SortedMap backed by sm.

Let's consider an example program to illustrate Collections algorithms.

Example

```
import java.util.*;

public class CollectionAlgorithmsExample {

    public static void main(String[] args) {

        ArrayList list = new ArrayList();
        PriorityQueue queue = new PriorityQueue();
        HashSet set = new HashSet();
        HashMap map = new HashMap();

        Random num = new Random();

        for(int i = 0; i < 5; i++) {
            list.add(num.nextInt(100));
            queue.add(num.nextInt(100));
            set.add(num.nextInt(100));
        }
    }
}
```

```

        map.put(i, num.nextInt(100));
    }

    System.out.println("List => " + list);
    System.out.println("Queue => " + queue);
    System.out.println("Set => " + set);
    System.out.println("Map => " + map);
    System.out.println("-----");

    Collections.sort(list);
    System.out.println("List in ascending order => " + list);

    System.out.println("Largest element in set => " + Collections.max(set));

    System.out.println("Smallest element in queue => " + Collections.min(queue));

    Collections.reverse(list);
    System.out.println("List in reverse order => " + list);

    Collections.shuffle(list);
    System.out.println("List after shuffle => " + list);
}
}

```

Legacy Class in Java :

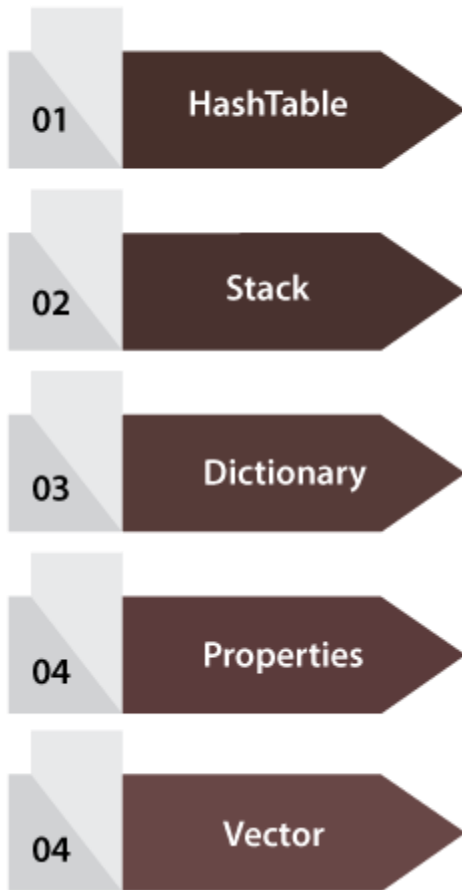
In the past decade, the **Collection** framework didn't include in Java. In the early version of Java, we have several classes and interfaces which allow us to store objects. After adding the Collection framework in **JSE 1.2**, for supporting the collections framework, these classes were re-engineered. So, classes and interfaces that formed the collections framework in the older version of Java are known as **Legacy classes**. For supporting generic in JDK5, these classes were re-engineered.

All the legacy classes are synchronized. The **java.util** package defines the following **legacy** classes:

1. HashTable
2. Stack

3. Dictionary
4. Properties
5. Vector

Legacy Classes



Vector Class

Vector is a special type of ArrayList that defines a dynamic array. ArrayList is not synchronized while **vector** is synchronized. The vector class has several legacy methods that are not present in the collection framework. Vector implements Iterable after the release of JDK 5 that defines the vector is fully compatible with collections, and vector elements can be iterated by the for-each loop.

Vector class provides the following four constructors:

44.7M
993
OOps Concepts in Java

1) **Vector()**

It is used when we want to create a default vector having the initial size of 10.

2) **Vector(int size)**

It is used to create a vector of specified capacity. It accepts size as a parameter to specify the initial capacity.

3) Vector(int size, int incr)

It is used to create a vector of specified capacity. It accepts two parameters size and increment parameters to specify the initial capacity and the number of elements to allocate each time when a vector is resized for the addition of objects.

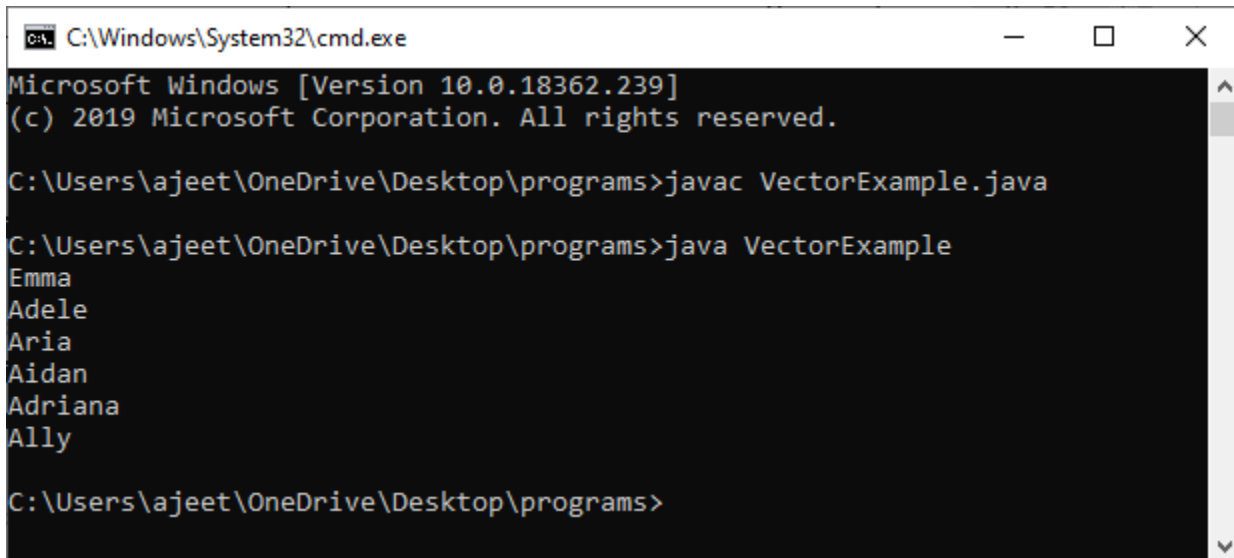
4) Vector(Collection c)

It is used to create a vector with the same elements which are present in the collection. It accepts the collection as a parameter.

VectorExample.java

```
import java.util.*;
public class VectorExample
{
    public static void main(String[] args)
    {
        Vector<String> vec = new Vector<String>();
        vec.add("Emma");
        vec.add("Adele");
        vec.add("Aria");
        vec.add("Aidan");
        vec.add("Adriana");
        vec.add("Ally");
        Enumeration<String> data = vec.elements();
        while(data.hasMoreElements())
        {
            System.out.println(data.nextElement());
        }
    }
}
```

Output:

A screenshot of a Windows Command Prompt window. The title bar shows 'C:\Windows\System32\cmd.exe'. The window content shows the following text:

```
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\ajeet\OneDrive\Desktop\programs>javac VectorExample.java

C:\Users\ajeet\OneDrive\Desktop\programs>java VectorExample
Emma
Adele
Aria
Aidan
Adriana
Ally

C:\Users\ajeet\OneDrive\Desktop\programs>
```

Hashtable Class:

The Hashtable class is similar to HashMap. It also contains the data into key/value pairs. It doesn't allow to enter any null key and value because it is synchronized. Just like Vector, Hashtable also has the following four constructors.

1) Hashtable()

It is used when we need to create a default HashTable having size 11.

2) Hashtable(int size)

It is used to create a HashTable of the specified size. It accepts size as a parameter to specify the initial size of it.

3) Hashtable(int size, float fillratio)

It creates the **Hashtable** of the specified size and fillratio. It accepts two parameters, size (of type int) and fillratio (of type float). The fillratio must be between 0.0 and 1.0. The **fillratio** parameter determines how full the hash table can be before it is resized upward. It means when we enter more elements than its capacity or size than the Hashtable is expended by multiplying its size with the fullratio.

4) Hashtable(Map< ? extends K, ? extends V> m)

It is used to create a Hashtable. The Hashtable is initialized with the elements present in **m**. The capacity of the Hashtable is the twice the number elements present in m.

HashtableExample.java

```
import java.util.*;
class HashtableExample
{
    public static void main(String args[])
    {
```

```

Hashtable<Integer,String> student = new Hashtable<Integer, String>();
student.put(new Integer(101), "Emma");
student.put(new Integer(102), "Adele");
student.put(new Integer(103), "Aria");
student.put(new Integer(104), "Ally");
Set dataset = student.entrySet();
Iterator iterate = dataset.iterator();
while(iterate.hasNext())
{
    Map.Entry map=(Map.Entry)iterate.next();
    System.out.println(map.getKey()+" "+map.getValue());
}
}
}

```

Output:

```

C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac HashtableExample.java
Note: HashtableExample.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\Users\ajeet\OneDrive\Desktop\programs>java HashtableExample
104 Ally
103 Aria
102 Adele
101 Emma

C:\Users\ajeet\OneDrive\Desktop\programs>_

```

Properties Class

Properties class extends Hashtable class to maintain the list of values. The list has both the key and the value of type string. The **Property** class has the following two constructors:

1) Properties()

It is used to create a Properties object without having default values.

2) Properties(Properties propdefault)

It is used to create the Properties object using the specified parameter of properties type for its default value.

The main difference between the Hashtable and Properties class is that in Hashtable, we cannot set a default value so that we can use it when no value is associated with a certain key. But in the Properties class, we can set the default value.

PropertiesExample.java

```

import java.util.*;
public class PropertiesExample
{
    public static void main(String[] args)
    {
        Properties prop_data = new Properties();
        prop_data.put("India", "Movies.");
        prop_data.put("United State", "Nobel Laureates and Getting Killed by Lawnmowers.");
        prop_data.put("Pakistan", "Field Hockey.");
        prop_data.put("China", "CO2 Emissions and Renewable Energy.");
        prop_data.put("Sri Lanka", "Cinnamon.");
        Set< ?> set_data = prop_data.keySet();
        for(Object obj: set_data)
        {
            System.out.println(obj+" is famous for "+ prop_data.getProperty((String)obj));
        }
    }
}

```

Output:

```

C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac PropertiesExample.java

C:\Users\ajeet\OneDrive\Desktop\programs>java PropertiesExample
Pakistan is famous for Field Hockey.
Sri Lanka is famous for Cinnamon.
China is famous for CO2 Emissions and Renewable Energy.
India is famous for Movies.
United State is famous for Nobel Laureates and Getting Killed by Lawnmowers.

C:\Users\ajeet\OneDrive\Desktop\programs>_

```

Stack Class

Stack class extends Vector class, which follows the LIFO(LAST IN FIRST OUT) principal for its elements. The stack implementation has only one default constructor, i.e., Stack().

1) Stack()

It is used to create a stack without having any elements.

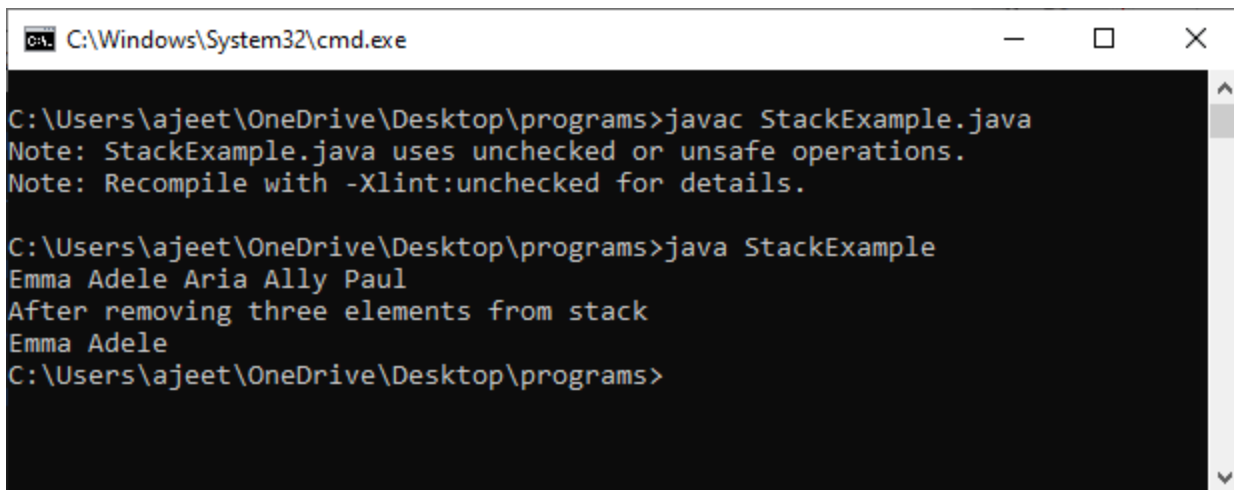
There are the following methods can be used with Stack class:

1. The **push()** method is used to add an object to the stack. It adds an element at the top of the stack.
2. The **pop()** method is used to get or remove the top element of the stack.
3. The **peek()** method is similar to the pop() method, but it can't remove the stack's top element using it.
4. The **empty()** method is used to check the emptiness of the stack. It returns true when the stack has no elements in it.
5. The **search()** method is used to ensure whether the specified object exists on the stack or not.

StackExample.java

```
import java.util.*;
class StackExample {
    public static void main(String args[]) {
        Stack stack = new Stack();
        stack.push("Emma");
        stack.push("Adele");
        stack.push("Aria");
        stack.push("Ally");
        stack.push("Paul");
        Enumeration enum1 = stack.elements();
        while(enum1.hasMoreElements())
            System.out.print(enum1.nextElement()+" ");
        stack.pop();
        stack.pop();
        stack.pop();
        System.out.println("\nAfter removing three elements from stack");
        Enumeration enum2 = stack.elements();
        while(enum2.hasMoreElements())
            System.out.print(enum2.nextElement()+" ");
    }
}
```

Output:



```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac StackExample.java
Note: StackExample.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\Users\ajeet\OneDrive\Desktop\programs>java StackExample
Emma Adele Aria Ally Paul
After removing three elements from stack
Emma Adele
C:\Users\ajeet\OneDrive\Desktop\programs>
```

Dictionary Class

The **Dictionary** class operates much like Map and represents the **key/value** storage repository. The **Dictionary** class is an abstract class that stores the data into the key/value pair. We can define the dictionary as a list of key/value pairs.

The dictionary class provides the following methods:

1. The **put(K key, V value)** method is used to add a key-value pair to the dictionary.
2. The **elements()** method is used to get the value representation in the dictionary.
3. The **get(Object key)** method is used to get the value mapped with the argumented key in the dictionary.
4. The **isEmpty()** method is used to check whether the dictionary is empty or not.
5. The **keys()** method is used to get the key representation in the dictionary.
6. The **remove(Object key)** method removes the data from the dictionary.
7. The **size()** method is used to get the size of the dictionary.

DictionaryExample.java

```
import java.util.*;

public class DictionaryExample
{
    public static void main(String[] args)
    {
        // Initializing Dictionary object
        Dictionary student = new Hashtable();

        // Using put() method to add elements
        student.put("101", "Emma");
        student.put("102", "Adele");
        student.put("103", "Aria");
        student.put("104", "Ally");
        student.put("105", "Paul");
    }
}
```

```

//Using the elements() method
for (Enumeration enum1 = student.elements(); enum1.hasMoreElements();
{
    System.out.println("The data present in the dictionary : " + enum1.nextElement());
}

// Using the get() method
System.out.println("\nName of the student 101 : " + student.get("101"));
System.out.println("Name of the student 102 : " + student.get("102"));

//Using the isEmpty() method
System.out.println("\n Is student dictionary empty? : " + student.isEmpty() + "\n");

// Using the keys() method
for (Enumeration enum2 = student.keys(); enum2.hasMoreElements();
{
    System.out.println("Ids of students: " + enum2.nextElement());
}

// Using the remove() method
System.out.println("\nDelete : " + student.remove("103"));
System.out.println("The name of the deleted student : " + student.get("123"));

System.out.println("\nThe size of the student dictionary is : " + student.size());

}
}

```

Output:

```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac DictionaryExample.java
Note: DictionaryExample.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\Users\ajeet\OneDrive\Desktop\programs>java DictionaryExample
The data present in the dictionary : Paul
The data present in the dictionary : Ally
The data present in the dictionary : Aria
The data present in the dictionary : Adele
The data present in the dictionary : Emma

Name of the student 101 : Emma
Name of the student 102 : Adele

Is student dictionary empty? : false

Ids of students: 105
Ids of students: 104
Ids of students: 103
Ids of students: 102
Ids of students: 101

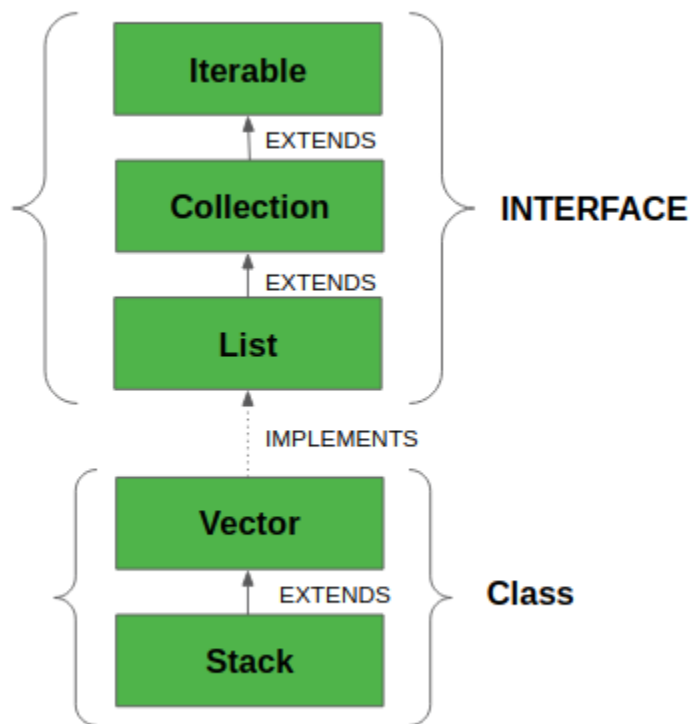
Delete : Aria
The name of the deleted student : null

The size of the student dictionary is : 4

C:\Users\ajeet\OneDrive\Desktop\programs>
```

Stack Class in Java

Java Collection framework provides a Stack class that models and implements a **Stack data structure**. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search, and peek. The class can also be said to extend Vector and treats the class as a stack with the five mentioned functions. The class can also be referred to as the subclass of Vector. The below diagram shows the **hierarchy of the Stack class**:



The class supports one *default constructor* **Stack()** which is used to *create an empty stack*.

Declaration:

```
public class Stack<E> extends Vector<E>
```

All Implemented Interfaces:

- **Serializable:** It is a marker interface that classes must implement if they are to be serialized and deserialized.
- **Cloneable:** This is an interface in Java which needs to be implemented by a class to allow its objects to be cloned.
- **Iterable<E>:** This interface represents a collection of objects which is iterable — meaning which can be iterated.
- **Collection<E>:** A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired.
- **List<E>:** The List interface provides a way to store the ordered collection. It is a child interface of Collection.
- **RandomAccess:** This is a marker interface used by List implementations to indicate that they support fast (generally constant time) random access.

How to Create a Stack?

In order to create a stack, we must import **java.util.stack** package and use the Stack() constructor of this class. The below example creates an empty Stack.

```
Stack<E> stack = new Stack<E>();
```

Here E is the type of Object.

Example:

- Java

// Java code for stack implementation

```
import java.io.*;
```

```
import java.util.*;
```

```
class Test
```

```
{
```

```
    // Pushing element on the top of the stack
```

```
    static void stack_push(Stack<Integer> stack)
```

```
    {
```

```
        for(int i = 0; i < 5; i++)
```

```
        {
```

```
            stack.push(i);
```

```
        }
```

```
    }
```

```
    // Popping element from the top of the stack
```

```
    static void stack_pop(Stack<Integer> stack)
```

```
    {
```

```
        System.out.println("Pop Operation:");
```

```
        for(int i = 0; i < 5; i++)
```

```

{
    Integer y = (Integer) stack.pop();

    System.out.println(y);
}

}

// Displaying element on the top of the stack

static void stack_peek(Stack<Integer> stack)
{
    Integer element = (Integer) stack.peek();

    System.out.println("Element on stack top: " + element);
}

// Searching element in the stack

static void stack_search(Stack<Integer> stack, int element)
{
    Integer pos = (Integer) stack.search(element);

    if(pos == -1)

        System.out.println("Element not found");

    else

        System.out.println("Element is found at position: " + pos);
}

```

```
}
```

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

```
{
```

```
    Stack<Integer> stack = new Stack<Integer>();
```

```
    stack_push(stack);
```

```
    stack_pop(stack);
```

```
    stack_push(stack);
```

```
    stack_peek(stack);
```

```
    stack_search(stack, 2);
```

```
    stack_search(stack, 6);
```

```
}
```

```
}
```

Output:

Pop Operation:

4

3

2

1

0

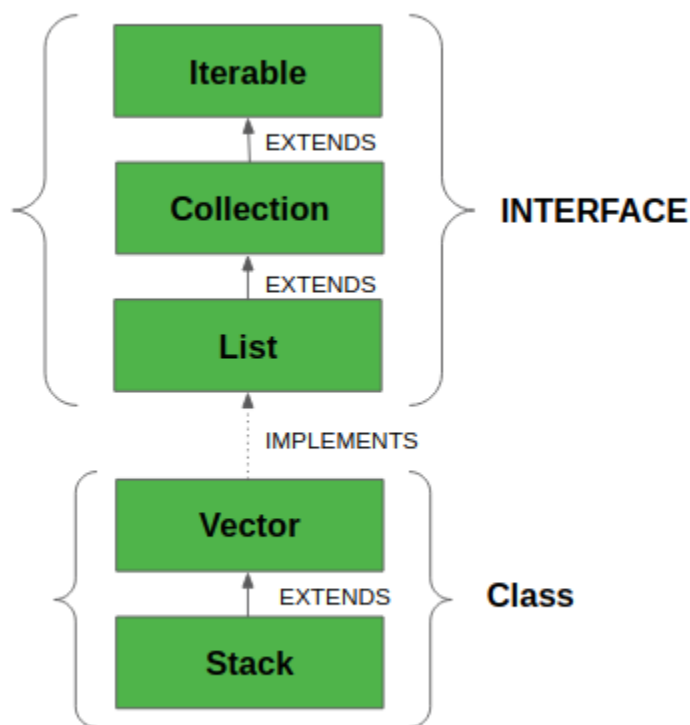
Element on stack top: 4

Element is found at position: 3

Element not found

Vector Class in Java

The Vector class implements a growable array of objects. Vectors fall in legacy classes, but now it is fully compatible with collections. It is found in `java.util` package and implement the `List` interface, so we can use all the methods of List interface as shown below as follows:



- Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index.
- They are very similar to `ArrayList`, but Vector is synchronized and has some legacy methods that the collection framework does not contain.
- It also maintains an insertion order like an `ArrayList`. Still, it is rarely used in a non-thread environment as it is **synchronized**, and due to this, it gives a poor performance in adding, searching, deleting, and updating its elements.
- The Iterators returned by the Vector class are fail-fast. In the case of concurrent modification, it fails and throws the **ConcurrentModificationException**.

Syntax:

```
public class Vector<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable
```

Here, **E** is the type of element.

- It extends `AbstractList` and implements `List` interfaces.
- It implements `Serializable`, `Cloneable`, `Iterable<E>`, `Collection<E>`, `List<E>`, `RandomAccess` interfaces.
- The directly known subclass is `Stack`.

Important points regarding the Increment of vector capacity are as follows:

If the increment is specified, Vector will expand according to it in each allocation cycle. Still, if the increment is not specified, then the vector's capacity gets doubled in each allocation cycle. Vector defines three protected data members:

- **int capacityIncrement:** Contains the increment value.
- **int elementCount:** Number of elements currently in vector stored in it.
- **Object elementData[]:** Array that holds the vector is stored in it.

Common Errors in the declaration of Vectors are as follows:

- Vector throws an **IllegalArgumentException** if the InitialSize of the vector defined is negative.
- If the specified collection is null, It throws **NullPointerException**.

Constructors

1. Vector(): Creates a default vector of the initial capacity is 10.

```
Vector<E> v = new Vector<E>();
```

2. Vector(int size): Creates a vector whose initial capacity is specified by size.

```
Vector<E> v = new Vector<E>(int size);
```

3. Vector(int size, int incr): Creates a vector whose initial capacity is specified by size and increment is specified by incr. It specifies the number of elements to allocate each time a vector is resized upward.

```
Vector<E> v = new Vector<E>(int size, int incr);
```

4. Vector(Collection c): Creates a vector that contains the elements of collection c.

```
Vector<E> v = new Vector<E>(Collection c);
```

StringTokenizer in Java

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc.

Constructors of StringTokenizer class

There are 3 constructors defined in the StringTokenizer class.

Java BitSet Class

The Java **BitSet** class implements a vector of bits. The BitSet grows automatically as more bits are needed. The BitSet class comes under *java.util* package. The BitSet class extends the Object class and provides the implementation of Serializable and Cloneable interfaces.

Each component of bit set contains at least one Boolean value. The contents of one BitSet may be changed by other BitSet using logical AND, logical OR and logical exclusive OR operations. The index of bits of BitSet class is represented by positive integers.

Each element of bits contains either true or false value. Initially, all bits of a set have the false value. A BitSet is not safe for multithreaded use without using external synchronization.

Date class in Java (With Examples)

The class Date represents a specific instant in time, with millisecond precision. The Date class of java.util package implements Serializable, Cloneable and Comparable interface. It provides constructors and methods to deal with date and time with java.

Constructors :

- **Date()** : Creates date object representing current date and time.
- **Date(long milliseconds)** : Creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.
- **Date(int year, int month, int date)**
- **Date(int year, int month, int date, int hrs, int min)**
- **Date(int year, int month, int date, int hrs, int min, int sec)**
- **Date(String s)**

Note : The last 4 constructors of the Date class are Deprecated.

// Java program to demonstrate constructors of Date

```
import java.util.*;
```

```
public class Main
```

```
{
```

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

```
    {
```

```
        Date d1 = new Date();
```

```
        System.out.println("Current date is " + d1);
```

```
        Date d2 = new Date(2323223232L);
```

```
        System.out.println("Date represented is "+ d2 );
```

```
    }
```

```
}
```

Output:

Current date is Tue Jul 12 18:35:37 IST 2016

Date represented is Wed Jan 28 02:50:23 IST 1970

Calendar Class in Java with examples:

Calendar class in Java is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable, Serializable, Cloneable interfaces.

As it is an Abstract class, so we cannot use a constructor to create an instance. Instead, we will have to use the static method Calendar.getInstance() to instantiate and implement a sub-class.

- Calendar.getInstance(): return a Calendar instance based on the current time in the default time zone with the default locale.
- Calendar.getInstance(TimeZone zone)
- Calendar.getInstance(Locale aLocale)
- Calendar.getInstance(TimeZone zone, Locale aLocale)

Java program to demonstrate getInstance() method:

```
// Date getTime(): It is used to return a
```

```
// Date object representing this
```

```
// Calendar's time value.
```

```
import java.util.*;
```

```
public class Calendar1 {
```

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

```
{
```

```
    Calendar c = Calendar.getInstance();
```

```
    System.out.println("The Current Date is:" + c.getTime());
```

```
}
```

```
}
```

Output:

The Current Date is:Tue Aug 28 11:10:40 UTC 2018

Generating random numbers in Java :

Java provides three ways to generate random numbers using some built-in methods and classes as listed below:

- **java.util.Random** class
- **Math.random** method : Can Generate Random Numbers of double type.
- **ThreadLocalRandom** class

1) java.util.Random

- For using this class to generate random numbers, we have to first create an instance of this class and then invoke methods such as `nextInt()`, `nextDouble()`, `nextLong()` etc using that instance.
- We can generate random numbers of types integers, float, double, long, booleans using this class.
- We can pass arguments to the methods for placing an upper bound on the range of the numbers to be generated. For example, `nextInt(6)` will generate numbers in the range 0 to 5 both inclusive.

```
// A Java program to demonstrate random number generation
```

```
// using java.util.Random;
```

```
import java.util.Random;
```

```
public class generateRandom{
```

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

```
{
```

```
    // create instance of Random class
```

```
    Random rand = new Random();
```

```
    // Generate random integers in range 0 to 999
```

```
    int rand_int1 = rand.nextInt(1000);
```

```
    int rand_int2 = rand.nextInt(1000);
```

```

        // Print random integers

        System.out.println("Random Integers: "+rand_int1);

        System.out.println("Random Integers: "+rand_int2);


        // Generate Random doubles

        double rand_dub1 = rand.nextDouble();

        double rand_dub2 = rand.nextDouble();


        // Print random doubles

        System.out.println("Random Doubles: "+rand_dub1);

        System.out.println("Random Doubles: "+rand_dub2);

    }

}

```

- Output:
- Random Integers: 547
- Random Integers: 126
- Random Doubles: 0.8369779739988428
- Random Doubles: 0.5497554388209912

Java formatter:

Java Formatter is a utility class that can make life simple when working with formatting stream output in Java. It is built to operate similarly to the C/C++ *printf* function. It is used to format and output data to a specific destination, such as a string or a file output stream. This article explores the class and illustrate some of its utility in everyday programming in [Java](#).

A Few Quick Examples

Using argument_index

```
Formatter f=new Formatter();  
f.format("%3$s %2$s %1$s", "fear",  
    "strengthen", "weakness");  
System.out.println(f);
```

Regionalize Output

```
StringBuilder builder=new StringBuilder();  
  
Formatter f=new Formatter(builder);  
f.format(Locale.FRANCE,"% .5f", -1325.789);  
System.out.println(f);  
  
Formatter f2=new Formatter();  
f2.format(Locale.CANADA, "% .5f", -1325.789);  
System.out.println(f2);
```

Regionalize Date

```
Formatter f3=new Formatter();  
f3.format(Locale.FRENCH,"% 1$te % 1$tB, % 1$tY",  
    Calendar.getInstance());  
System.out.println(f3);
```

```
Formatter f4=new Formatter();  
f4.format(Locale.GERMANY,"%1$te %1$tB, %1$tY",  
    Calendar.getInstance());  
System.out.println(f4);
```

Using %n and %% Specifiers

```
Formatter f = new Formatter();  
f.format("Format%n %.2f%% complete", 46.6);  
System.out.println(f);
```

Scanner Class in Java

Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc and strings. It is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.

- To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
- To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort()
- To read strings, we use nextLine().
- To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) function returns the first character in that string.

Let us look at the code snippet to read data of various data types.

// Java program to read data of various types using Scanner class.

```
import java.util.Scanner;  
  
public class ScannerDemo1  
{
```

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

{

    // Declare the object and initialize with

    // predefined standard input object

    Scanner sc = new Scanner(System.in);


    // String input

    String name = sc.nextLine();


    // Character input

    char gender = sc.next().charAt(0);


    // Numerical data input

    // byte, short and float can be read

    // using similar-named functions.

    int age = sc.nextInt();

    long mobileNo = sc.nextLong();

    double cgpa = sc.nextDouble();


    // Print the values to check if the input was correctly obtained.

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

    System.out.println("Gender: "+gender);
```

```
System.out.println("Age: "+age);

System.out.println("Mobile Number: "+mobileNo);

System.out.println("CGPA: "+cgpa);

}

}
```

Input :

Geek

F

40

9876543210

9.9

Output :

Name: Geek

Gender: F

Age: 40

Mobile Number: 9876543210

CGPA: 9.9