UNIT 4
Collections in Java

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

The Java **collections** framework provides a set of interfaces and classes to implement various data structures and algorithms.

For example, the `LinkedList` class of the collections framework provides the implementation of the doubly-linked list data structure.

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

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

What is Collection in Java

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

What is a framework in Java

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

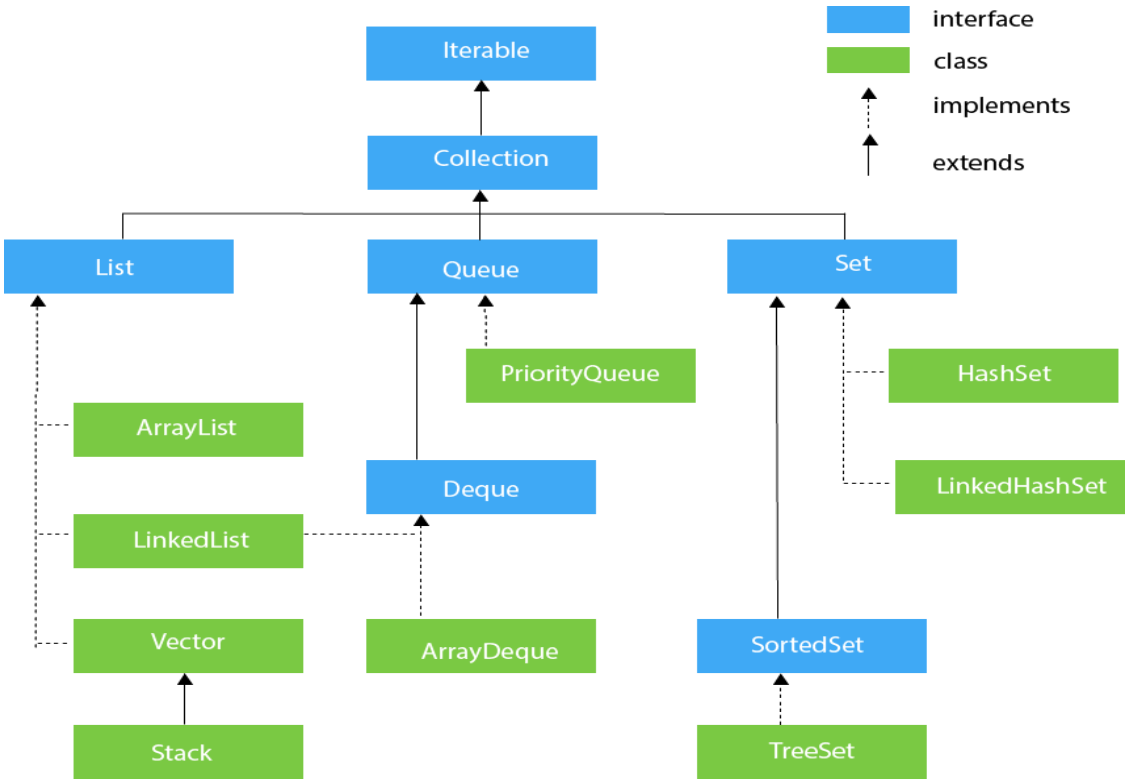
What is Collection framework

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

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

Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.



3. Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public booleanaddAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public booleanremoveAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	default booleanremoveIf(Predicate<? super E> filter)	It is used to delete all the elements of the collection that satisfy the specified predicate.
6	public booleanretainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.

7	public int size()	It returns the total number of elements in the collection.
8	public void clear()	It removes the total number of elements from the collection.
9	public boolean contains(Object element)	It is used to search an element.
10	public booleancontainsAll(Collection<?> c)	It is used to search the specified collection in the collection.
11	public Iterator iterator()	It returns an iterator.
12	public Object[] toArray()	It converts collection into array.
13	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
14	public booleanisEmpty()	It checks if collection is empty.
15	default Stream<E>parallelStream()	It returns a possibly parallel Stream with the collection as its source.
16	default Stream<E> stream()	It returns a sequential Stream with the collection as its source.
17	default Spliterator<E>spliterator()	It generates a Spliterator over the specified elements in the collection.
18	public boolean equals(Object element)	It matches two collections.
19	public inthashCode()	It returns the hash code number of the collection.

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	It returns true if the iterator has more elements

	booleanhasNext()	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.,

1. terator<T> iterator()

It returns the iterator over the elements of type T.

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 (Objectobj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

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.

1. List <data-type> list2 = **new** LinkedList();
2. List <data-type> list3 = **new** Vector();

List <data-type> list4 = **new** Stack To instantiate the List interface, we must use :

3. List <data-type> list1= **new** ArrayList();
4. ();

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

The classes that implement the List interface are given below.

ArrayList

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

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr
.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

LinkedList

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

Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
```

```
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ravi
Ajay
```

Vector

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

Consider the following example.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String args[]){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Amit");
v.add("Ashish");
v.add("Garima");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ayush
Amit
Ashish
Garima
```

Stack

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

Consider the following example.

```
import java.util.*;
public class TestJavaCollection4{
public static void main(String args[]){
Stack<String> stack = new Stack<String>();
stack.push("Ayush");
stack.push("Garvit");
stack.push("Amit");
stack.push("Ashish");
stack.push("Garima");
stack.pop();
Iterator<String> itr=stack.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ayush
Garvit
Amit
Ashish
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

1. `Queue<String> q1 = new PriorityQueue();`
2. `Queue<String> q2 = new ArrayDeque();`

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit Sharma");
queue.add("Vijay Raj");
queue.add("JaiShankar");
queue.add("Raj");
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 Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
```

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

1. Deque d = new ArrayDeque();

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection6{
public static void main(String[] args) {
//Creating Deque and adding elements
Deque<String> deque = new ArrayDeque<String>();
deque.add("Gautam");
deque.add("Karan");
deque.add("Ajay");
//Traversing elements
for (String str : deque) {
System.out.println(str);
}
}
}
```

Output:

```
Gautam
Karan
Ajay
```

Set Interface

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

Set can be instantiated as:

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

HashSet

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

Consider the following example.

```
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Vijay
Ravi
Ajay
```

LinkedHashSet

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

Consider the following example.

```
import java.util.*;
public class TestJavaCollection8{
public static void main(String args[]){
```

```
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ajay
```

SortedSet Interface

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

The SortedSet can be instantiated as:

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

TreeSet

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

Consider the following example:

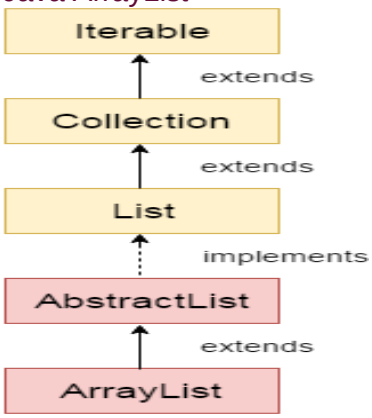
```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
```

```
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ajay
Ravi
Vijay
```

Java ArrayList



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 the List interface here. The ArrayList maintains the insertion order internally.

It inherits the AbstractList class and implements List interface.

The important points about the 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 the array works on an index basis.
- In ArrayList, manipulation is a 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.

- We can not create an array list of the primitive types, such as int, float, char, etc. It is required to use the required wrapper class in such cases. For example:

```
1. ArrayList<int> al = ArrayList<int>(); // does not work
2. ArrayList<Integer> al = new ArrayList<Integer>(); // works fine
```

- Java ArrayList gets initialized by the size. The size is dynamic in the array list, which varies according to the elements getting added or removed from the list.

Hierarchy of ArrayList class

As shown in the above diagram, the Java ArrayList class extends AbstractList class which implements the List interface. The List interface extends the Collection and Iterable interfaces in hierarchical order.

ArrayList class declaration

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

```
1. public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, S
   erializable
```

Constructors of ArrayList

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection<? extends E> 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.

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(intrequiredCapacity)	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.
booleanisEmpty()	It returns true if the list is empty, otherwise false.
Iterator()	
listIterator()	
intlastIndexOf(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.
intindexOf(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.
booleanremoveIf(Predicate<? super E> filter)	It is used to remove all the elements from the list that satisfies the given predicate.

protected void removeRange(intfromIndex, inttoIndex)	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 the specified comparator.
Spliterator<E>spliterator()	It is used to create a spliterator over the elements in a list.
List<E>subList(intfromIndex, inttoIndex)	It is used to fetch all the elements that 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 a Java collection.

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

Let's see the new generic example of creating java collection.

```
1. 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 object in it. If you try to add another type of object, it gives a *compile-time error*.

For more information on Java generics, click here Java Generics Tutorial.

Java ArrayList Example

FileName: ArrayListExample1.java

```
import java.util.*;
 public class ArrayListExample1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
      list.add("Mango");//Adding object in arraylist
      list.add("Apple");
      list.add("Banana");
      list.add("Grapes");
      //Printing the arraylist object
      System.out.println(list);
 }
}
```

Output:

```
[Mango, Apple, Banana, Grapes]
```

Iterating ArrayList using Iterator

Let's see an example to traverse ArrayList elements using the Iterator interface.

FileName: ArrayListExample2.java

```
import java.util.*;
public class ArrayListExample2{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
  list.add("Mango");//Adding object in arraylist
  list.add("Apple");
  list.add("Banana");
  list.add("Grapes");
  //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 move to next
  }
 }
}
```

Output:

```
Mango
```

```
Apple
Banana
Grapes
```

Iterating ArrayList using For-each loop

Let's see an example to traverse the ArrayList elements using the for-each loop

FileName: ArrayListExample3.java

```
import java.util.*;
public class ArrayListExample3{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
  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);

 }
}
```

Output:

```
Mango
Apple
Banana
Grapes
```

Get and Set ArrayList

The *get() method* returns the element at the specified index, whereas the *set() method* changes the element.

FileName: ArrayListExample4.java

```
import java.util.*;
public class ArrayListExample4{
 public static void main(String args[]){
  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 2nd element, because index starts
   from 0
    //changing the element
    al.set(1,"Dates");
    //Traversing list
    for(String fruit:al)
      System.out.println(fruit);


   }
  }
```

Output:

```
Returning element: Apple
Mango
Dates
Banana
Grapes
```

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.

FileName: SortArrayList.java

```
  import java.util.*;
  class SortArrayList{
   public static void main(String args[]){
    //Creating a list of fruits
    List<String> list1=new ArrayList<String>();
    list1.add("Mango");
    list1.add("Apple");
    list1.add("Banana");
    list1.add("Grapes");
    //Sorting the list
    Collections.sort(list1);
     //Traversing list through the for-each loop
    for(String fruit:list1)
      System.out.println(fruit);
```

```
    System.out.println("Sorting numbers...");
     //Creating a list of numbers
     List<Integer> list2=new ArrayList<Integer>();
     list2.add(21);
     list2.add(11);
     list2.add(51);
     list2.add(1);
     //Sorting the list
     Collections.sort(list2);
      //Traversing list through the for-each loop
     for(Integer number:list2)
       System.out.println(number);
   }


  }
```

Output:

```
Apple
Banana
Grapes
Mango
Sorting numbers...
1
11
21
51
```

Ways to iterate the elements of the collection in Java

There are various ways to traverse the collection elements:

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

Iterating Collection through remaining ways

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

FileName: ArrayList4.java

```
import java.util.*;
class ArrayList4{
 public static void main(String args[]){
   ArrayList<String> list=new ArrayList<String>();//Creating arraylist
         list.add("Ravi");//Adding object in arraylist
         list.add("Vijay");
         list.add("Ravi");
         list.add("Ajay");

         System.out.println("Traversing list through List Iterator:");
         //Here, element iterates in reverse order
           ListIterator<String> list1=list.listIterator(list.size());
           while(list1.hasPrevious())
           {
              String str=list1.previous();
              System.out.println(str);
           }
       System.out.println("Traversing list through for loop:");
         for(int i=0;i<list.size();i++)
         {
          System.out.println(list.get(i));
         }

       System.out.println("Traversing list through forEach() method:");
       //The forEach() method is a new feature, introduced in Java 8.
          list.forEach(a->{ //Here, we are using lambda expression
             System.out.println(a);
           });

          System.out.println("Traversing list through forEachRemaining() method:");
           Iterator<String> itr=list.iterator();
           itr.forEachRemaining(a-> //Here, we are using lambda expression
           {
          System.out.println(a);
           });
 }
}
```

Output:

```
Traversing list through List Iterator:
Ajay
Ravi
Vijay
Ravi
Traversing list through for loop:
Ravi
Vijay
Ravi
Ajay
Traversing list through forEach() method:
Ravi
Vijay
Ravi
Ajay
Traversing list through forEachRemaining() method:
Ravi
Vijay
Ravi
Ajay
```

User-defined class objects in *Java ArrayList*

Let's see an example where we are storing Student class object in an array list.

FileName: ArrayList5.java

```
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 ArrayList5{
 public static void main(String args[]){
  //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);
    }
   }
  }
```

Output:

```
101 Sonoo 23
102 Ravi 21
103 Hanumat 25
```

Java ArrayList Serialization and Deserialization Example

Let's see an example to serialize an ArrayList object and then deserialize it.

FileName: ArrayList6.java

```
import java.io.*;
import java.util.*;
 class ArrayList6 {

      public static void main(String [] args)
      {
        ArrayList<String> al=new ArrayList<String>();
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ajay");

        try
        {
           //Serialization
           FileOutputStream fos=new FileOutputStream("file");
```

```
            ObjectOutputStream oos=new ObjectOutputStream(fos);
            oos.writeObject(al);
            fos.close();
            oos.close();
            //Deserialization
            FileInputStream fis=new FileInputStream("file");
            ObjectInputStream ois=new ObjectInputStream(fis);
          ArrayList  list=(ArrayList)ois.readObject();
          System.out.println(list);
         }catch(Exception e)
         {
            System.out.println(e);
         }
      }
    }
```

Output:

```
[Ravi, Vijay, Ajay]
```

Java ArrayList example to add elements

Here, we see different ways to add an element.

FileName: ArrayList7.java

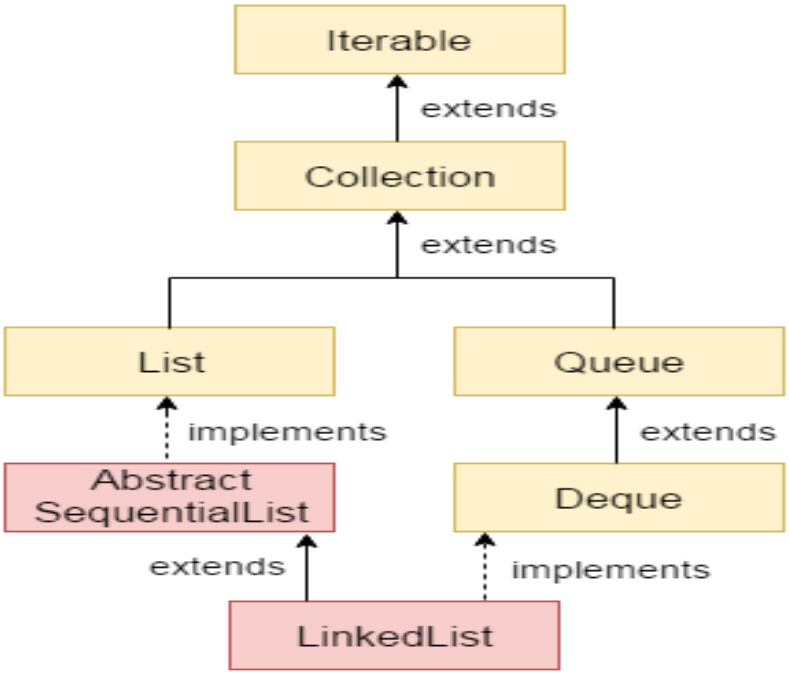
```
import java.util.*;
 class ArrayList7{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
         System.out.println("Initial list of elements: "+al);
         //Adding elements to the end of the list
         al.add("Ravi");
         al.add("Vijay");
         al.add("Ajay");
         System.out.println("After invoking add(E e) method: "+al);
         //Adding an element at the specific position
         al.add(1, "Gaurav");
         System.out.println("After invoking add(int index, E element) method: "+al);
         ArrayList<String> al2=new ArrayList<String>();
         al2.add("Sonoo");
```

```
        al2.add("Hanumat");
        //Adding second list elements to the first list
        al.addAll(al2);
        System.out.println("After invoking addAll(Collection<? extends E> c) method: "+al);
        ArrayList<String> al3=new ArrayList<String>();
        al3.add("John");
        al3.add("Rahul");
        //Adding second list elements to the first list at specific position
        al.addAll(1, al3);
        System.out.println("After invoking addAll(int index, Collection<? extends E> c) method: "+al);

    }
}
```

Output:

```
Initial list of elements: []
After invoking add(E e) method: [Ravi, Vijay, Ajay]
After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]
After invoking addAll(Collection<? extends E> c) method:
[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addAll(int index, Collection<? extends E> c) method:
[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
```

Java 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.

fig- doubly linked list

LinkedList class declaration

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

1. **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.

Methods of Java LinkedList

Method	Description
boolean add(E e)	It is used to append the specified element to the end of a list.
void add(int index, E element)	It is used to insert the specified element at the specified position index in 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 addFirst(E e)	It is used to insert the given element at the beginning of a list.
void addLast(E e)	It is used to append the given element to the end of a list.
void clear()	It is used to remove all the elements from a list.
Object clone()	It is used to return a shallow copy of an ArrayList.
boolean contains(Object o)	It is used to return true if a list contains a specified element.
Iterator<E> descendingIterator()	It is used to return an iterator over the elements in a deque in reverse sequential order.
E element()	It is used to retrieve the first element of a list.
E get(int index)	It is used to return the element at the specified position in a list.
E getFirst()	It is used to return the first element in a list.
E getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.
ListIterator<E> listIterator(int index)	It is used to return a list-iterator of the elements in proper sequence, starting at the specified position in the list.
boolean offer(E e)	It adds the specified element as the last element of a list.
boolean offerFirst(E e)	It inserts the specified element at the front of a list.
boolean offerLast(E e)	It inserts the specified element at the end of a list.
E peek()	It retrieves the first element of a list
E peekFirst()	It retrieves the first element of a list or returns null if a list is empty.
E peekLast()	It retrieves the last element of a list or returns null if a list is empty.
E poll()	It retrieves and removes the first element of a list.
E pollFirst()	It retrieves and removes the first element of a list, or returns null if a list is empty.
E pollLast()	It retrieves and removes the last element of a list, or returns null if a list is empty.

E pop()	It pops an element from the stack represented by a list.
void push(E e)	It pushes an element onto the stack represented by a list.
E remove()	It is used to retrieve and removes the first element of a list.
E remove(int index)	It is used to remove the element at the specified position in a list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element in a list.
E removeFirst()	It removes and returns the first element from a list.
boolean removeFirstOccurrence(Object o)	It is used to remove the first occurrence of the specified element in a list (when traversing the list from head to tail).
E removeLast()	It removes and returns the last element from a list.
boolean removeLastOccurrence(Object o)	It removes the last occurrence of the specified element in a list (when traversing the list from head to tail).
E set(int index, E element)	It replaces the element at the specified position in a list with the specified element.
Object[] toArray()	It is used to return an array containing all the elements in a list in proper sequence (from first to the last element).
<T> T[] toArray(T[] a)	It returns an array containing all the elements in the proper sequence (from first to the last element); the runtime type of the returned array is that of the specified array.
int size()	It is used to return the number of elements in a list.

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 LinkedList example to add elements

Here, we see different ways to add elements.

```
import java.util.*;
public class LinkedList2{
 public static void main(String args[ ]){
 LinkedList<String> ll=new LinkedList<String>();
        System.out.println("Initial list of elements: "+ll);
        ll.add("Ravi");
        ll.add("Vijay");
        ll.add("Ajay");
        System.out.println("After invoking add(E e) method: "+ll);
        ll.add(1, "Gaurav");            //Adding an element at the specific position

        System.out.println("After invoking add(int index, E element) method: "+ll);
        LinkedList<String> ll2=new LinkedList<String>();
        ll2.add("Sonoo");
        ll2.add("Hanumat");

        ll.addAll(ll2);    //Adding second list elements to the first list

        System.out.println("After invoking addAll(Collection<? extends E> c) method: "+ll);
        LinkedList<String> ll3=new LinkedList<String>();
        ll3.add("John");
        ll3.add("Rahul");

         ll.addAll(1, ll3);   //Adding second list elements to the first list at specific position

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

        ll.addFirst("Lokesh");            //Adding an element at the first position

        System.out.println("After invoking addFirst(E e) method: "+ll);
        //Adding an element at the last position
        ll.addLast("Harsh");
        System.out.println("After invoking addLast(E e) method: "+ll);

 }
}
```

```
Initial list of elements: []
After invoking add(E e) method: [Ravi, Vijay, Ajay]
After invoking add(int index, E element) method: [Ravi, Gaurav, Vijay, Ajay]
After invoking addAll(Collection<? extends E> c) method:
[Ravi, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addAll(int index, Collection<? extends E> c) method:
```

```
[Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addFirst(E e) method:
[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat]
After invoking addLast(E e) method:
[Lokesh, Ravi, John, Rahul, Gaurav, Vijay, Ajay, Sonoo, Hanumat, Harsh]
```

Java LinkedList example to remove elements

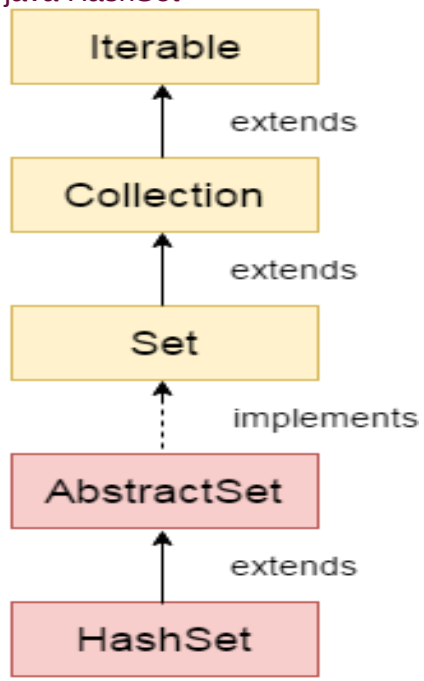
Here, we see different ways to remove an element.

```
import java.util.*;
public class LinkedList3 {

  public static void main(String [] args)
  {
    LinkedList<String> ll=new LinkedList<String>();
    ll.add("Ravi");
    ll.add("Vijay");
    ll.add("Ajay");
    ll.add("Anuj");
    ll.add("Gaurav");
    ll.add("Harsh");
    ll.add("Virat");
    ll.add("Gaurav");
    ll.add("Harsh");
    ll.add("Amit");
    System.out.println("Initial list of elements: "+ll);
  //Removing specific element from arraylist
      ll.remove("Vijay");
      System.out.println("After invoking remove(object) method: "+ll);
  //Removing element on the basis of specific position
      ll.remove(0);
      System.out.println("After invoking remove(index) method: "+ll);
      LinkedList<String> ll2=new LinkedList<String>();
      ll2.add("Ravi");
      ll2.add("Hanumat");
  // Adding new elements to arraylist
      ll.addAll(ll2);
      System.out.println("Updated list : "+ll);
  //Removing all the new elements from arraylist
      ll.removeAll(ll2);
      System.out.println("After invoking removeAll() method: "+ll);
  //Removing first element from the list
      ll.removeFirst();
      System.out.println("After invoking removeFirst() method: "+ll);
   //Removing first element from the list
      ll.removeLast();
      System.out.println("After invoking removeLast() method: "+ll);
   //Removing first occurrence of element from the list
```

```
        ll.removeFirstOccurrence("Gaurav");
        System.out.println("After invoking removeFirstOccurrence() method: "+ll);
    //Removing last occurrence of element from the list
        ll.removeLastOccurrence("Harsh");
        System.out.println("After invoking removeLastOccurrence() method: "+ll);

        //Removing all the elements available in the list
        ll.clear();
        System.out.println("After invoking clear() method: "+ll);
  }
}
```

```
Initial list of elements: [Ravi, Vijay, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
After invoking remove(object) method: [Ravi, Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
After invoking remove(index) method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
Updated list : [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit, Ravi, Hanumat]
After invoking removeAll() method: [Ajay, Anuj, Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
After invoking removeFirst() method: [Gaurav, Harsh, Virat, Gaurav, Harsh, Amit]
After invoking removeLast() method: [Gaurav, Harsh, Virat, Gaurav, Harsh]
After invoking removeFirstOccurrence() method: [Harsh, Virat, Gaurav, Harsh]
After invoking removeLastOccurrence() method: [Harsh, Virat, Gaurav]
After invoking clear() method: []
```

java 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.
- HashSetis the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.

Hierarchy of HashSet class

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

HashSet class declaration

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

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

Constructors of Java HashSet class

SN	Constructor	Description
1)	HashSet( )	It is used to construct a default HashSet.
2)	HashSet(int capacity)	It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
3)	HashSet(int capacity, float loadFactor)	It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.
4)	HashSet(Collection<? extends E> c)	It is used to initialize the hash set by using the elements of the collection c.

Methods of Java HashSet class

Various methods of Java HashSet class are as follows:

SN	Modifier & Type	Method	Description
1)	Boolean	add(E e)	It is used to add the specified element to this set if it is not already present.
2)	Void	clear()	It is used to remove all of the elements from the set.
3)	Object	clone()	It is used to return a shallow copy of this

			HashSet instance: the elements themselves are not cloned.
4)	Boolean	contains(Object o)	It is used to return true if this set contains the specified element.
5)	Boolean	isEmpty()	It is used to return true if this set contains no elements.
6)	Iterator<E>	iterator()	It is used to return an iterator over the elements in this set.
7)	boolean	remove(Object o)	It is used to remove the specified element from this set if it is present.
8)	int	size()	It is used to return the number of elements in the set.
9)	Spliterator<E>	spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

Java HashSet Example

Let's see a simple example of HashSet. Notice, the elements iterate in an unordered collection.

```
import java.util.*;
class HashSet1{
 public static void main(String args[]){
  //Creating HashSet and adding elements
    HashSet<String> set=new HashSet( );
          set.add("One");
          set.add("Two");
          set.add("Three");
          set.add("Four");
          set.add("Five");
          Iterator<String> i=set.iterator();
          while(i.hasNext())
          {
          System.out.println(i.next());
          }
} }}
```

```
Five
One
Four
Two
Three
```

Java HashSet example ignoring duplicate elements

In this example, we see that HashSet doesn't allow duplicate elements.

```
  import java.util.*;
  class HashSet2{
   public static void main(String args[]){
```

```
  //Creating HashSet and adding elements
  HashSet<String> set=new HashSet<String>();
  set.add("Ravi");
  set.add("Vijay");
  set.add("Ravi");
  set.add("Ajay");
  //Traversing elements
  Iterator<String> itr=set.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

```
Ajay
Vijay
Ravi
```

Java HashSet example to remove elements

```
import java.util.*;
public class HashSetRemoveExample2 {
    public static void main(String[] args) {
        HashSet<Integer> hashSetObject = new HashSet ( );
        hashSetObject.add(45);
        hashSetObject.add(67);
        hashSetObject.add(98);
        hashSetObject.add(24);
        //Display HashSet
            System.out.println("HashSet: " + hashSetObject);
            //Remove elements using remove() method
            hashSetObject.remove(98);
            hashSetObject.remove(24);
            //Displaying the HashSet after removal
          System.out.println("HashSet after removing elements: " + hashSetObject);
    }
}
```

Output: Compile by: javac HashSetRemoveExample2.java

Run by: java HashSetRemoveExample2

```
HashSet: [98, 67, 45, 24]
HashSet after removing elements: [67, 45]
```

java HashSet contains() Method

The contains() method of Java HashSet class is used to check if this HashSet contains the specified element or not. It returns true if element is found otherwise, returns false.

Syntax

Following is the declaration of **contains()** method:

```
public boolean contains(Object o)
```

```
import java.util.*;
public class HashSetContainsExample1 {
    public static void main(String[] args) {
        //Create hash set
```

```
        HashSet<Integer> hset = new HashSet<Integer>();
        //Add elements to hash set
        hset.add(11);
        hset.add(21);
        hset.add(15);
        hset.add(110);
        hset.add(151);
        //Print HashSet elements
        System.out.println("Hash set Elements: "+ hset);
        //Check for "110" in the set
      System.out.println("Does the Set contains '110'? :- "+hset.contains(110));
      //Check if the Set contains "555"
      System.out.println("Does the Set contains '555'? :- "+hset.contains(555));
      }
  }
```

Output:

```
Hash set Elements: [21, 151, 11, 110, 15]
Does the Set contains '110'? :- true
Does the Set contains '555'? :- false
```

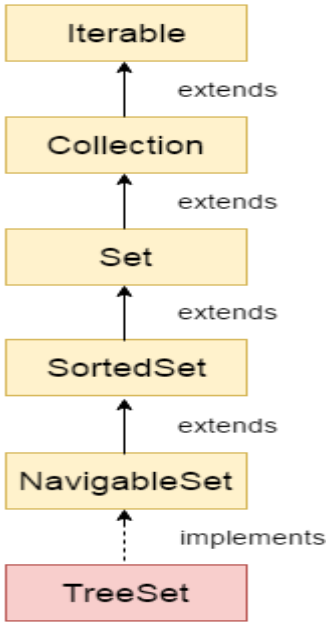
Example 2

```
  import java.util.*;
  public class HashSetContainsExample2 {
     public static void main(String[] args) {
        //Get HashMap object from method init()
        @SuppressWarnings("rawtypes")
        HashSet studentSet = init();

        System.out.print("Enter student name:");      //Ask input from console
        Scanner s = new Scanner(System.in);
        String name = s.nextLine();
        s.close();
        //Check if the console input is in the student database
        if(studentSet.contains(name)){
           System.out.println(name +" found on student list.");
        }
        else{
           System.out.println(name +" name not on the student list.");
        }
  }
  @SuppressWarnings("rawtypes")
  private static HashSet init() {
     //CreateHashSet object
     HashSet<String> studentSet = new HashSet<>();
     //Add values to HashMap
     studentSet.add("Rishi");
     studentSet.add("Sharon");
     studentSet.add("Pawan");
     return studentSet;
     }
  }
```

Output:

```
Enter student name: Rishi
Rishi found on student list.
```

Java TreeSet class



Tree set

1. Balanced tree
2. Duplicate not allowed
3. Insertion order not fallowed
4. Some Sorting technique is applicable
5. Herogeneous obj are not allowed(ClassCastException)
6. Null insertion accepted only once

Tree set constructors

1. **Treeset t= new Treeset( ) ;// default natural sorting order(assending)**
2. **Treeset t= new Treeset(comparator c); //costomised sorting order**
3. **Treeset t= new Treeset(collection c);**
4. **Treeset t= new Treeset(sortedset s);**

Example

```
import java.util.Treeset;
class T1{
 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");
al.add(new Integer(10)); ///classcastexception
al.add(null);    // nullpointerexception
```

```
  //Traversing elements
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
(Or)
 System.out.println(al);
  } } }
```

Output:

```
Ajay
Ravi
Vijay
```

Comparable interface:

This interface present in java.lang package it contains only one method CompareTo( )

Ex:

Obj1.compareTo(obj2)

- Return –ve iff obj1 has to come before obj2
- Return +ve iff obj1 has to come after obj2
- Return 0 iff obj1 &obj2 equal.

sytem.out.println("A" compareTo ("z")) ; -ve

if we depending on default natural sorting order internally JVM will call CompareTO() will inserting objectis to the Treeset. Hence the objects should be Comparable.

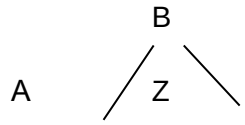
Treeset t =new Treeset();

t. add("B");

t.add("z") //"z compareto B ; +ve

t.add("A"); // " Z compreto B -ve

System.out.println(t);



Output: [A B Z]

-If we are not satisfied with default natural sorting order we can define our own customized sorting by using **Comparator**

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 the 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. &allow only when the set is empty.
- Java TreeSet class maintains ascending order.
- The TreeSet can only allow those generic types that are comparable. For example The Comparable interface is being implemented by the StringBuffer class.

Internal Working of The TreeSet Class

TreeSet is being implemented using a binary search tree, which is self-balancing just like a Red-Black Tree. Therefore, operations such as a search, remove, and add consume O(log(N)) time. The reason behind this is there in the self-balancing tree. It is there to ensure that the tree height never exceeds O(log(N)) for all of the mentioned operations. Therefore, it is one of the efficient data structures in order to keep the large data that is sorted and also to do operations on it.

Synchronization of The TreeSet Class

As already mentioned above, the TreeSet class is not synchronized. It means if more than one thread concurrently accesses a tree set, and one of the accessing threads modify it, then the synchronization must be done manually. It is usually done by doing some object synchronization that encapsulates the set. However, in the case where no such object is found, then the set must be wrapped with the help of the Collections.synchronizedSet() method. It is advised to use the method during creation time in order to avoid the unsynchronized access of the set. The following code snippet shows the same.

```
1. TreeSet treeSet = new TreeSet();
2. Set syncrSet = Collections.synchronziedSet(treeSet);
```

Hierarchy of TreeSet class

As shown in the above diagram, the 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.

```
1. public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializabl
   e
```

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.

Methods of Java TreeSet Class

Method	Description
boolean add(E e)	It is used to add the specified element to this set if it is not already present.
boolean addAll(Collection<? extends E> c)	It is used to add all of the elements in the specified collection to this set.

E ceiling(E e)	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
Comparator<? super E> comparator()	It returns a comparator that arranges elements in order.
Iterator descendingIterator()	It is used to iterate the elements in descending order.
NavigableSet descendingSet()	It returns the elements in reverse order.
E floor(E e)	It returns the equal or closest least element of the specified element from the set, or null there is no such element.
SortedSet headSet(E toElement)	It returns the group of elements that are less than the specified element.
NavigableSet headSet(E toElement, boolean inclusive)	It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element.
E higher(E e)	It returns the closest greatest element of the specified element from the set, or null there is no such element.
Iterator iterator()	It is used to iterate the elements in ascending order.
E lower(E e)	It returns the closest least element of the specified element from the set, or null there is no such element.
E pollFirst()	It is used to retrieve and remove the lowest(first) element.
E pollLast()	It is used to retrieve and remove the highest(last) element.
Spliterator spliterator()	It is used to create a late-binding and fail-fast spliterator over the elements.
NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	It returns a set of elements that lie between the given range.
SortedSet subSet(E fromElement, E toElement))	It returns a set of elements that lie between the given range which includes fromElement and excludes toElement.

SortedSet tailSet(E fromElement)	It returns a set of elements that are greater than or equal to the specified element.
NavigableSet tailSet(E fromElement, boolean inclusive)	It returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element.
boolean contains(Object o)	It returns true if this set contains the specified element.
boolean isEmpty()	It returns true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
void clear()	It is used to remove all of the elements from this set.
Object clone()	It returns a shallow copy of this TreeSet instance.
E first()	It returns the first (lowest) element currently in this sorted set.
E last()	It returns the last (highest) element currently in this sorted set.
int size()	It returns the number of elements in this set.

Java TreeSet Example 1:

Let's see a simple example of Java TreeSet.

FileName: TreeSet1.java

```
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.

FileName: TreeSet2.java

```
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
```

Java TreeSet Example 3:

Let's see an example to retrieve and remove the highest and lowest Value.

FileName: TreeSet3.java

```
        import java.util.*;
        class TreeSet3{
         public static void main(String args[]){
         TreeSet<Integer> set=new TreeSet<Integer>();
                set.add(24);
                set.add(66);
                set.add(12);
```

```
        set.add(15);
        System.out.println("Lowest Value: "+set.pollFirst());
        System.out.println("Highest Value: "+set.pollLast());
    }
}
```

Output:

```
Lowest Value: 12
Highest Value: 66
```

Java TreeSet Example 4:

In this example, we perform various NavigableSet operations.

FileName: TreeSet4.java

```
import java.util.*;
class TreeSet4{
 public static void main(String args[]){
  TreeSet<String> set=new TreeSet<String>();
        set.add("A");
        set.add("B");
        set.add("C");
        set.add("D");
        set.add("E");
        System.out.println("Initial Set: "+set);

        System.out.println("Reverse Set: "+set.descendingSet());

        System.out.println("Head Set: "+set.headSet("C", true));

        System.out.println("SubSet: "+set.subSet("A", false, "E", true));

        System.out.println("TailSet: "+set.tailSet("C", false));
 }
}
```

Output:

```
Initial Set: [A, B, C, D, E]
Reverse Set: [E, D, C, B, A]
Head Set: [A, B, C]
SubSet: [B, C, D, E]
TailSet: [D, E]
```

---

java Queue Interface

The interface Queue is available in the java.util package and does extend the Collection interface. It is used to keep the elements that are processed in the First In First Out (FIFO) manner. It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list.

Being an interface, the queue requires, for the declaration, a concrete class, and the most common classes are the LinkedList and PriorityQueue in Java. Implementations done by these classes are not thread safe. If it is required to have a thread safe implementation, PriorityBlockingQueue is an available option.

Queue Interface Declaration

1. **public interface** Queue<E> **extends** Collection<E>

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.

Features of a Queue

The following are some important features of a queue.

- As discussed earlier, FIFO concept is used for insertion and deletion of elements from a queue.
- The Java Queue provides support for all of the methods of the Collection interface including deletion, insertion, etc.
- PriorityQueue, ArrayBlockingQueue and LinkedList are the implementations that are used most frequently.
- The NullPointerException is raised, if any null operation is done on the BlockingQueues.
- Those Queues that are present in the *util* package are known as Unbounded Queues.
- Those Queues that are present in the *util.concurrent* package are known as bounded Queues.
- All Queues barring the Deques facilitates removal and insertion at the head and tail of the queue; respectively. In fact, deques support element insertion and removal at both ends.

PriorityQueue Class

PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority. It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue. However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.

PriorityQueue Class Declaration

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

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

Java PriorityQueue Example

FileName: TestCollection12.java

```
1. import java.util.*;
2. class TestCollection12{
3. public static void main(String args[]){
4. PriorityQueue<String> queue=new PriorityQueue<String>();
5. queue.add("Amit");
6. queue.add("Vijay");
7. queue.add("Karan");
8. queue.add("Jai");
9. queue.add("Rahul");
10. System.out.println("head:"+queue.element());
11. System.out.println("head:"+queue.peek());
12. System.out.println("iterating the queue elements:");
13. Iterator itr=queue.iterator();
14. while(itr.hasNext()){
15. System.out.println(itr.next());
16. }
17. queue.remove();
18. queue.poll();
19. System.out.println("after removing two elements:");
20. Iterator<String> itr2=queue.iterator();
21. while(itr2.hasNext()){
22. System.out.println(itr2.next());
23. }
24. }
25. }
```

Test it Now

Output:

```
head:Amit
head:Amit
iterating the queue elements:
Amit
Jai
Karan
Vijay
Rahul
after removing two elements:
Karan
Rahul
Vijay
```

java Deque Interface

The interface called Deque is present in java.util package. It is the subtype of the interface queue. The Deque supports the addition as well as the removal of elements from both ends of the data structure. Therefore, a deque can be used as a stack or a queue. We know that the stack supports the Last In First Out (LIFO) operation, and the operation First In First Out is supported by a queue. As a deque supports

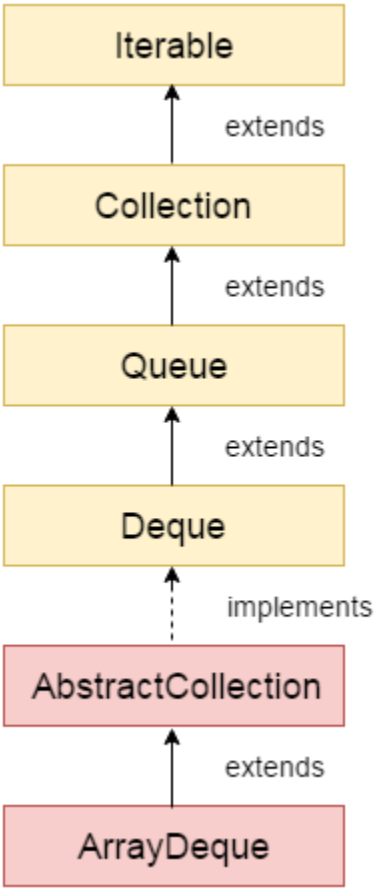
both, either of the mentioned operations can be performed on it. Deque is an acronym for **"double ended queue".**

Deque Interface declaration

1. **public interface** Deque<E> **extends** Queue<E>

Methods of Java Deque Interface

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 retrieve and removes the head of this deque.
Object poll()	It is used to retrieve and removes the head of this deque, or returns null if this deque is empty.
Object element()	It is used to retrieve, but does not remove, the head of this deque.
Object peek()	It is used to retrieve, but does not remove, the head of this deque, or returns null if this deque is empty.
Object peekFirst()	The method returns the head element of the deque. The method does not remove any element from the deque. Null is returned by this method, when the deque is empty.
Object peekLast()	The method returns the last element of the deque. The method does not remove any element from the deque. Null is returned by this method, when the deque is empty.
Boolean offerFirst(e)	Inserts the element e at the front of the queue. If the insertion is successful, true is returned; otherwise, false.
Object offerLast(e)	Inserts the element e at the tail of the queue. If the insertion is successful, true is returned; otherwise, false.



ArrayDeque class

We know that it is not possible to create an object of an interface in Java. Therefore, for instantiation, we need a class that implements the Deque interface, and that class is ArrayDeque. It grows and shrinks as per usage. It also inherits the AbstractCollection class.

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.

ArrayDeque class declaration

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

1. **public class** ArrayDeque<E> **extends** AbstractCollection<E> **implements** Deque<E>, Cloneable, Seriali zable

Java ArrayDeque Example

FileName: ArrayDequeExample.java

```
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
```

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

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:

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:

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:

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));
    }
  }
}
```

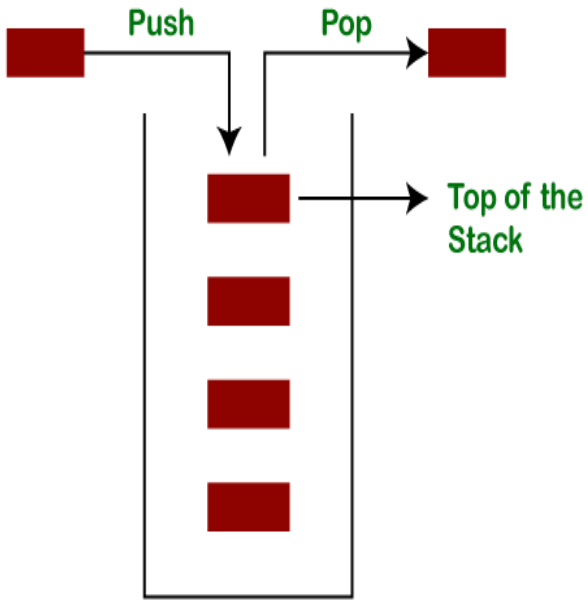
Output:

Java Stack

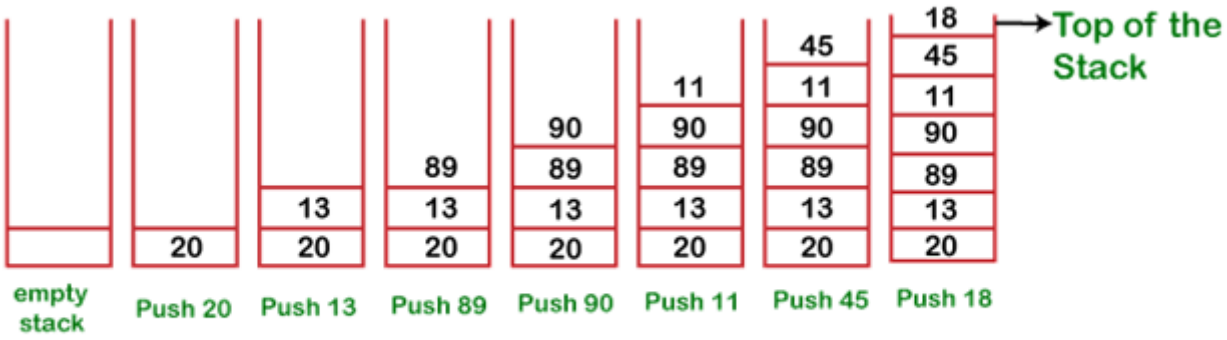
The **stack** is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out** (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the **Stack class** that provides different operations such as push, pop, search, etc.

In this section, we will discuss the **Java Stack class**, its **methods,** and **implement** the stack data structure in a Java program. But before moving to the Java Stack class have a quick view of how the stack works.

The stack data structure has the two most important operations that are **push** and **pop**. The push operation inserts an element into the stack and pop operation removes an element from the top of the stack. Let's see how they work on stack.

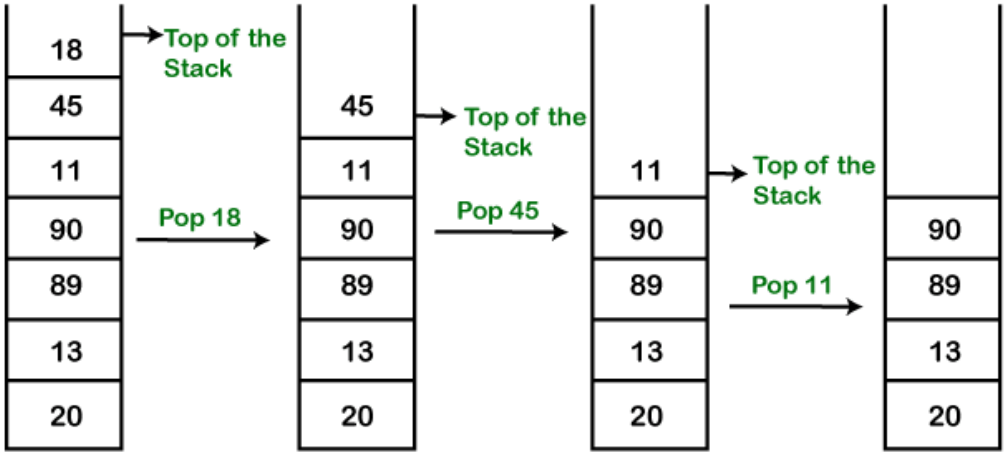


Let's push 20, 13, 89, 90, 11, 45, 18, respectively into the stack.



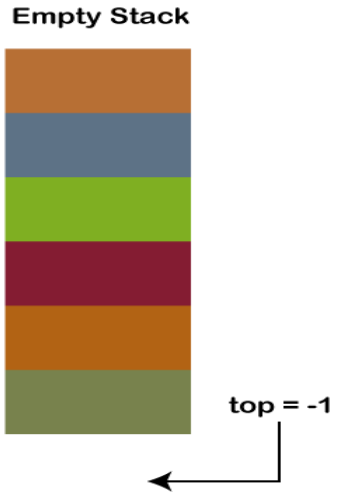
Push operation

Let's remove (pop) 18, 45, and 11 from the stack.



Pop operation

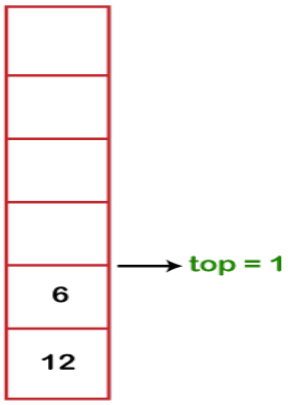
Empty Stack: If the stack has no element is known as an **empty stack**. When the stack is empty the value of the top variable is -1.



When we push an element into the stack the top is **increased by 1**. In the following figure,

- Push 12, top=0
- Push 6, top=1
- Push 9, top=2

When we pop an element from the stack the value of top is **decreased by 1**. In the following figure, we have popped 9.

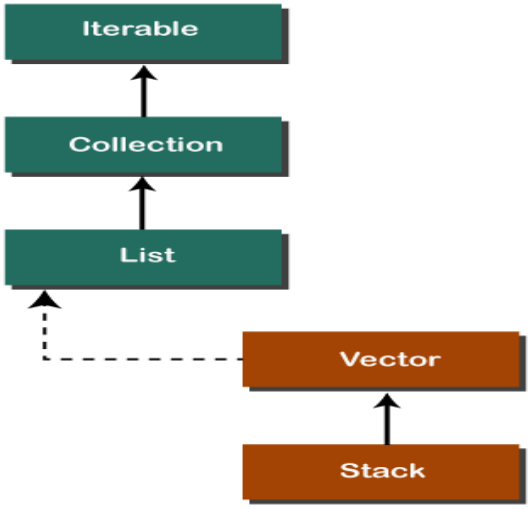


The following table shows the different values of the top.

Top value	Meaning
-1	It shows the stack is empty.
0	The stack has only an element.
N-1	The stack is full.
N	The stack is overflow.

Java Stack Class

In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class. It also implements interfaces **List, Collection, Iterable, Cloneable, Serializable.** It represents the LIFO stack of objects. Before using the Stack class, we must import the java.util package. The stack class arranged in the Collections framework hierarchy, as shown below.



Stack Class Constructor

The Stack class contains only the **default constructor** that creates an empty stack.

1. **public** Stack()

Creating a Stack

If we want to create a stack, first, import the java.util package and create an object of the Stack class.

```
1. Stack stk = new Stack();
```

Or

```
1. Stack<type> stk = new Stack<>();
```

Where type denotes the type of stack like Integer, String, etc.

Methods of the Stack Class

We can perform push, pop, peek and search operation on the stack. The Java Stack class provides mainly five methods to perform these operations. Along with this, it also provides all the methods of the Java Vector class.

Method	Modifier and Type	Method Description
empty()	boolean	The method checks the stack is empty or not.
push(E item)	E	The method pushes (insert) an element onto the top of the stack.
pop()	E	The method removes an element from the top of the stack and returns the same element as the value of that function.
peek()	E	The method looks at the top element of the stack without removing it.
search(Object o)	int	The method searches the specified object and returns the position of the object.

Stack Class empty() Method

The **empty()** method of the Stack class check the stack is empty or not. If the stack is empty, it returns true, else returns false. We can also use the isEmpty() method of the Vector class.

Syntax

```
1. public boolean empty()
```

Returns: The method returns true if the stack is empty, else returns false.

In the following example, we have created an instance of the Stack class. After that, we have invoked the empty() method two times. The first time it returns **true** because we have not pushed any element into the stack. After that, we have pushed elements into the stack. Again we have invoked the empty() method that returns **false** because the stack is not empty.

StackEmptyMethodExample.java

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

```
//creating an instance of Stack class
Stack<Integer> stk= new Stack<>();
// checking stack is empty or not
boolean result = stk.empty();
System.out.println("Is the stack empty? " + result);
// pushing elements into stack
stk.push(78);
stk.push(113);
stk.push(90);
stk.push(120);
//prints elements of the stack
System.out.println("Elements in Stack: " + stk);
result = stk.empty();
System.out.println("Is the stack empty? " + result);
}
}
```

Output:

```
Is the stack empty? true
Elements in Stack: [78, 113, 90, 120]
Is the stack empty? false
```

Iterator in Java

In Java, an **Iterator** is one of the Java cursors. **Java Iterator** is an interface that is practiced in order to iterate over a collection of Java object components entirety one by one. It is free to use in the Java programming language since the Java 1.2 Collection framework. It belongs to java.util package.

Though Java Iterator was introduced in Java 1.2, however, it is still not the oldest tool available to traverse through the elements of the Collection object. The oldest Iterator in the Java programming language is the Enumerator predated Iterator. Java Iterator interface succeeds the enumerator iterator that was practiced in the beginning to traverse over some accessible collections like the ArrayLists.

The Java Iterator is also known as the **universal cursor** of Java as it is appropriate for all the classes of the Collection framework. The Java Iterator also helps in the operations like READ and REMOVE. When we compare the Java Iterator interface with the enumeration iterator interface, we can say that the names of the methods available in Java Iterator are more precise and straightforward to use.

Advantages of Java Iterator

Iterator in Java became very prevalent due to its numerous advantages. The advantages of Java Iterator are given as follows -

Backward Skip 10sPlay VideoForward Skip 10s

- The user can apply these iterators to any of the classes of the Collection framework.
- In Java Iterator, we can use both of the read and remove operations.
- If a user is working with a for loop, they cannot modernize(add/remove) the Collection, whereas, if they use the Java Iterator, they can simply update the Collection.

- The Java Iterator is considered the Universal Cursor for the Collection API.
- The method names in the Java Iterator are very easy and are very simple to use.

Disadvantages of Java Iterator

Despite the numerous advantages, the Java Iterator has various disadvantages also. The disadvantages of the Java Iterator are given below -

- The Java Iterator only preserves the iteration in the forward direction. In simple words, the Java Iterator is a uni-directional Iterator.
- The replacement and extension of a new component are not approved by the Java Iterator.
- In CRUD Operations, the Java Iterator does not hold the various operations like CREATE and UPDATE.
- In comparison with the Spliterator, Java Iterator does not support traversing elements in the parallel pattern which implies that Java Iterator supports only Sequential iteration.
- In comparison with the Spliterator, Java Iterator does not support more reliable execution to traverse the bulk volume of data.

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
```

for-each loop

for-each version of for loop can also be used for traversing the elements of a collection. But this can only be used if we don't want to modify the contents of a collection and we don't want any **reverse** access. for-each loop can cycle through any collection of object that implements Iterable interface.

Exmaple:

```
import java.util.*;

class Demo

{

  public static void main(String[] args)

  {

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

    ls.add("a");

    ls.add("b");

    ls.add("c");

    ls.add("d");

    for(String str : ls)

    {

      System.out.print(str+" ");

    }

  }

}
```

```
a b c d
```

Points to Remember

- The Java Iterator is an interface added in the Java Programming language in the Java 1.2 Collection framework. It belongs to java.util package.
- It is one of the Java Cursors that are practiced to traverse the objects of the collection framework.
- The Java Iterator is used to iterate the components of the collection object one by one.
- The Java Iterator is also known as the Universal cursor of Java as it is appropriate for all the classes of the Collection framework.
- The Java Iterator also supports the operations like READ and REMOVE.
- The methods names of the Iterator class are very simple and easy to use compared to the method names of Enumeration Iterator.

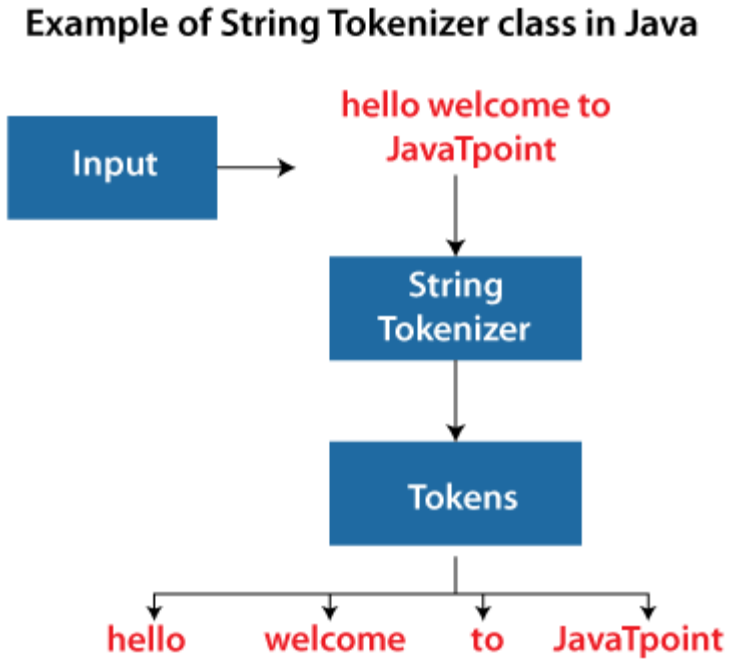
StringTokenizer in Java

1. StringTokenizer
2. Methods of StringTokenizer
3. Example of StringTokenizer

The **java.util.StringTokenizer** class allows you to break a String into tokens. It is simple way to break a String. It is a legacy class of Java.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class. We will discuss about the StreamTokenizer class in I/O chapter.

In the StringTokenizer class, the delimiters can be provided at the time of creation or one by one to the tokens.



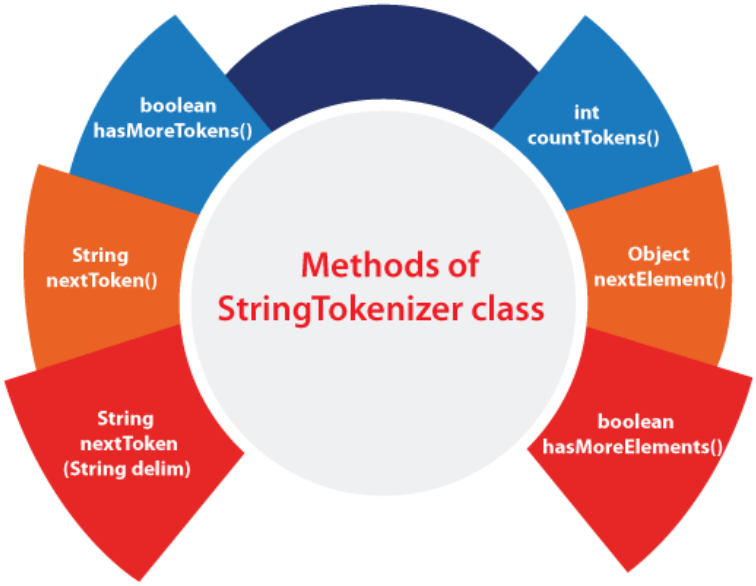
Constructors of the StringTokenizer Class

There are 3 constructors defined in the StringTokenizer class.

Constructor	Description
StringTokenizer(String str)	It creates StringTokenizer with specified string.
StringTokenizer(String str, String delim)	It creates StringTokenizer with specified string and delimiter.
StringTokenizer(String str, String delim, boolean returnValue)	It creates StringTokenizer with specified string, delimiter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

Methods of the StringTokenizer Class

The six useful methods of the StringTokenizer class are as follows:



Methods	Description
boolean hasMoreTokens()	It checks if there is more tokens available.
String nextToken()	It returns the next token from the StringTokenizer object.
String nextToken(String delim)	It returns the next token based on the delimiter.
boolean hasMoreElements()	It is the same as hasMoreTokens() method.
Object nextElement()	It is the same as nextToken() but its return type is Object.

int countTokens()	It returns the total number of tokens.

Example of StringTokenizer Class

Let's see an example of the StringTokenizer class that tokenizes a string "my name is khan" on the basis of whitespace.

Simple.java

```
import java.util.StringTokenizer;
public class Simple{
 public static void main(String args[]){
   StringTokenizer st = new StringTokenizer("my name is khan"," ");
    while (st.hasMoreTokens()) {
        System.out.println(st.nextToken());
    }
   }
}
```

Output:

```
my
name
is
khan
```

The above Java code, demonstrates the use of StringTokenizer class and its methods hasMoreTokens() and nextToken().

Example of nextToken(String delim) method of the StringTokenizer class

Test.java

```
import java.util.*;

public class Test {
  public static void main(String[] args) {
     StringTokenizer st = new StringTokenizer("my,name,is,khan");

    // printing next token
    System.out.println("Next token is : " + st.nextToken(","));
  }
}
```

Output:

```
Next token is : my
```

Java Calendar Class

Java Calendar class 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 interface.

Java Calendar class declaration

Let's see the declaration of java.util.Calendar class.

1. **public abstract class** Calendar **extends** Object
2. **implements** Serializable, Cloneable, Comparable<Calendar>

List of Calendar Methods

No	Method	Description
1.	public void add(int field, int amount)	Adds the specified (signed) amount of time to the given calendar field.
2.	public boolean after (Object when)	The method Returns true if the time represented by this Calendar is after the time represented by when Object.
3.	public boolean before(Object when)	The method Returns true if the time represented by this Calendar is before the time represented by when Object.
4.	public final void clear(int field)	Set the given calendar field value and the time value of this Calendar undefined.
5.	public Object clone()	Clone method provides the copy of the current object.
6.	public int compareTo(Calendar anotherCalendar)	The compareTo() method of Calendar class compares the time values (millisecond offsets) between two calendar object.
7.	protected void complete()	It fills any unset fields in the calendar fields.
8.	protected abstract void computeFields()	It converts the current millisecond time value time to calendar field values in fields[].
9.	protected abstract void computeTime()	It converts the current calendar field values in fields[] to the millisecond time value time.
10.	public boolean equals(Object object)	The equals() method compares two objects for equality and Returns true if they are equal.
11.	public int get(int field)	In get() method fields of the calendar are passed as the parameter, and this method Returns the value of fields passed as the parameter.

12.	public int getActualMaximum(int field)	Returns the Maximum possible value of the calendar field passed as the parameter to getActualMaximum() method.
13.	public int getActualMinimum(int field)	Returns the Minimum possible value of the calendar field passed as parameter to getActualMinimum() methot.
14.	public static Set<String> getAvailableCalendarTypes()	Returns a set which contains string set of all available calendar type supported by Java Runtime Environment.
15.	public static Locale[] getAvailableLocales()	Returns an array of all locales available in java runtime environment.
16.	public String getCalendarType()	Returns in string all available calendar type supported by Java Runtime Environment.
17.	public String getDisplayName(int field, int style, Locale locale)	Returns the String representation of the calendar field value passed as the parameter in a given style and local.
18.	public Map<String,Integer> getDisplayNames(int field, int style, Locale locale)	Returns Map representation of the calendar field value passed as the parameter in a given style and local.
19.	public int getFirstDayOfWeek()	Returns the first day of the week in integer form.

Java Calendar Class Example

```
import java.util.Calendar;
public class CalendarExample1 {
  public static void main(String[] args) {
  Calendar calendar = Calendar.getInstance();
  System.out.println("The current date is : " + calendar.getTime());
  calendar.add(Calendar.DATE, -15);
  System.out.println("15 days ago: " + calendar.getTime());
  calendar.add(Calendar.MONTH, 4);
  System.out.println("4 months later: " + calendar.getTime());
  calendar.add(Calendar.YEAR, 2);
  System.out.println("2 years later: " + calendar.getTime());
  }
}
```

Test it Now

Output:

```
The current date is : Thu Jan 19 18:47:02 IST 2017
15 days ago: Wed Jan 04 18:47:02 IST 2017
4 months later: Thu May 04 18:47:02 IST 2017
2 years later: Sat May 04 18:47:02 IST 2019
```

Formatter class in java

The **Formatter** is a built-in class in java used for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output in java programming. The Formatter class is defined as final class inside the **java.util** package.

The Formatter class implements **Cloneable** and **Flushable** interface.

The Formatter class in java has the following constructors.

S.No.	Constructor with Description
1	**Formatter()** It creates a new formatter.
2	**Formatter(Appendable a)** It creates a new formatter with the specified destination.
3	**Formatter(Appendable a, Locale l)** It creates a new formatter with the specified destination and locale.
4	**Formatter(File file)** It creates a new formatter with the specified file.
5	**Formatter(File file, String charset)** It creates a new formatter with the specified file and charset.
6	**Formatter(File file, String charset, Locale l)**

Example

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

```
public class FormatterClassExample {

        public static void main(String[] args) {

                Formatter formatter=new Formatter();
                formatter.format("%2$5s %1$5s %3$5s", "Smart", "BTech", "Class");
                System.out.println(formatter);

                formatter = new Formatter();
                formatter.format(Locale.FRANCE,"%.5f", -1325.789);
                System.out.println(formatter);


            String name = "Java";
            formatter = new Formatter();
            formatter.format(Locale.US,"Hello %s !", name);
            System.out.println("" + formatter + " " + formatter.locale());

            formatter = new Formatter();
            formatter.format("%.4f", 123.1234567);
            System.out.println("Decimal floating-point notation to 4 places: " + formatter);

            formatter = new Formatter();
            formatter.format("%010d", 88);
            System.out.println("value in 10 digits: " + formatter);
        }
```