Collection in Java

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.

1. Iterator<T> iterator()

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.

Collection interface builds the foundation on which the collection framework depends.

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

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.

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

ArrayList

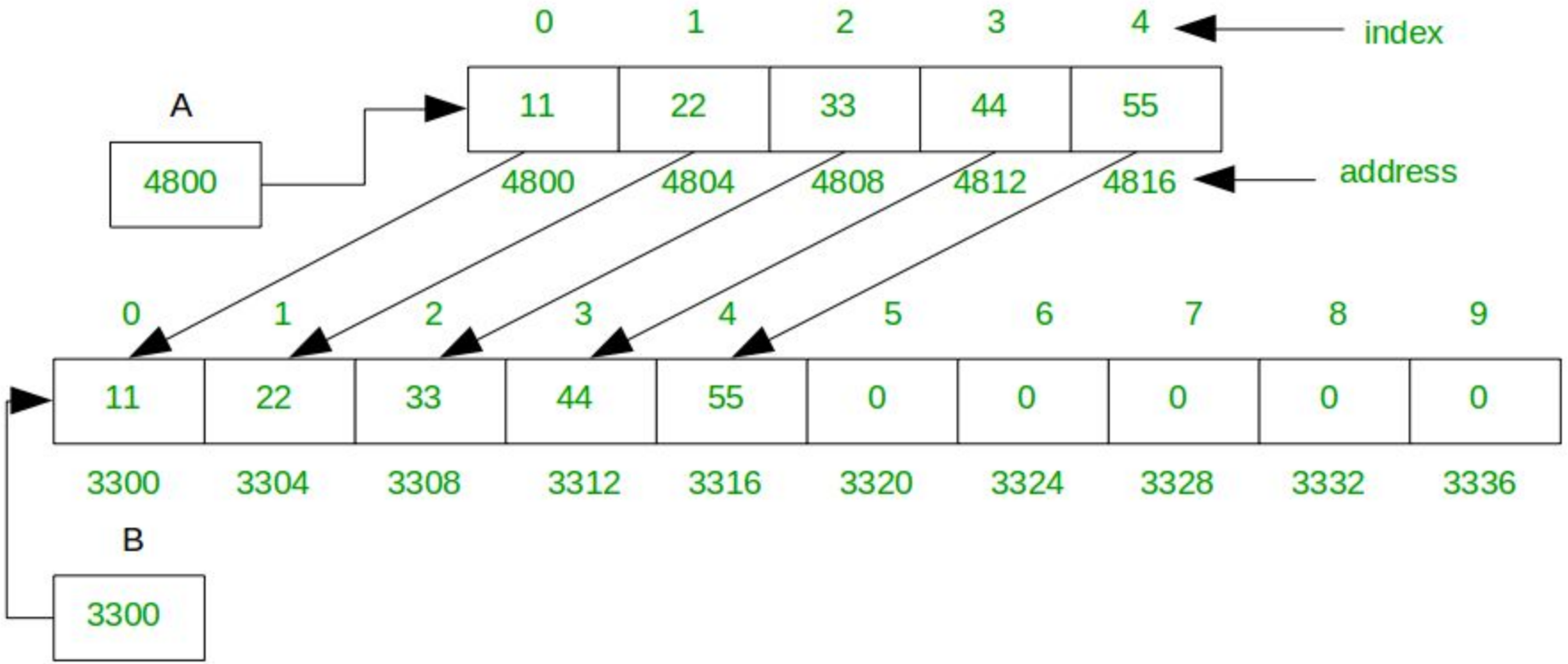
The ArrayList class implements the List interface.

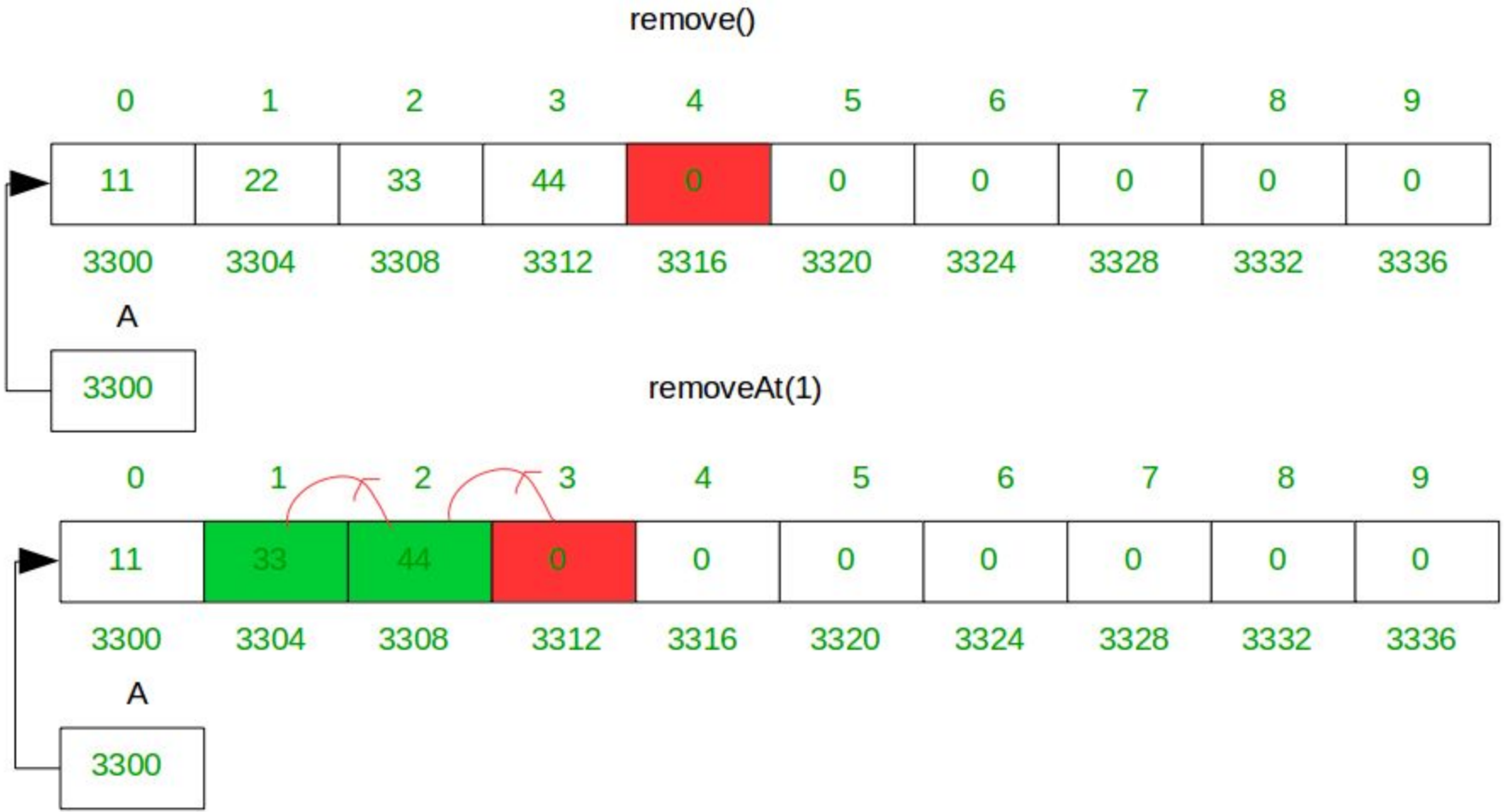
It uses a dynamic array to store the duplicate element of different data types.

The ArrayList class maintains the insertion order and is non-synchronized.

Non-Synchronized means that two or more threads can access the methods of that particular class at any given time

The elements stored in the ArrayList class can be randomly accessed.


0
1
2
3
4
index
A
11
22
33
44
55
4800
4800
4804
4808
4812
4816
address
0
1
2
3
4
5
6
7
8
9
11
22
33
44
55
0
0
0
0
0
3300
3304
3308
3312
3316
3320
3324
3328
3332
3336
B
3300


remove()
0 1 2 3 4 5 6 7 8 9
11 22 33 44 0 0 0 0 0 0
3300 3304 3308 3312 3316 3320 3324 3328 3332 3336
A
3300
removeAt(1)
0 1 2 3 4 5 6 7 8 9
11 33 44 0 0 0 0 0 0 0
3300 3304 3308 3312 3316 3320 3324 3328 3332 3336
A
3300

```
1.  import java.util.*;
2.  class TestJavaCollection1{
3.  public static void main(String args[]){
4.  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5.  list.add("Ravi");//Adding object in arraylist
6.  list.add("Vijay");
7.  list.add("Ravi");
8.  list.add("Ajay");
9.  //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. while(itr.hasNext()){
12. System.out.println(itr.next());
13. }
14. }
15. }
```

Ravi
Vijay
Ravi
Ajay

LinkedList

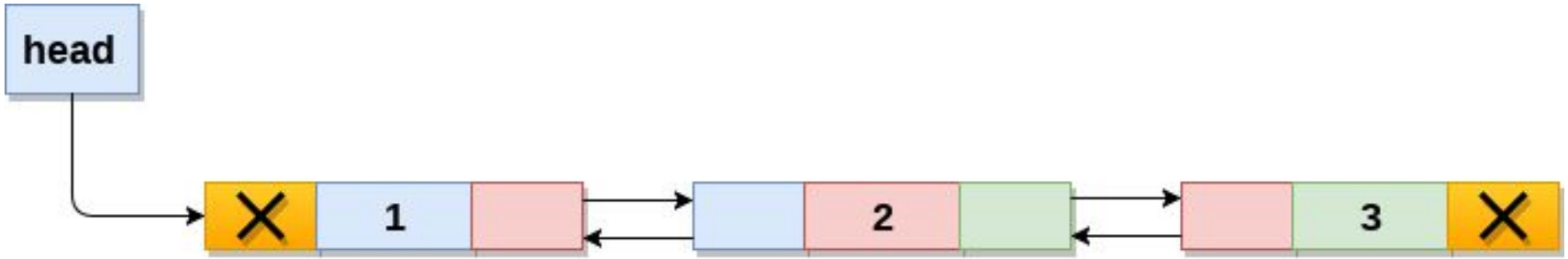
LinkedList implements the Collection interface.

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.



Doubly Linked List

```
1.  import java.util.*;
2.  public class TestJavaCollection2{
3.  public static void main(String args[]){
4.  LinkedList<String> al=new LinkedList<String>();
5.  al.add("Ravi");
6.  al.add("Vijay");
7.  al.add("Ravi");
8.  al.add("Ajay");
9.  Iterator<String> itr=al.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

```
Ravi
Vijay
Ravi
Ajay
```

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

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. Enumeration and Iterator

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.

Java Synchronization is better option where we want to allow only one thread to access the shared resource

```
1.  import java.util.*;
2.  public class TestJavaCollection3{
3.  public static void main(String args[]){
4.  Vector<String> v=new Vector<String>();
5.  v.add("Ayush");
6.  v.add("Amit");
7.  v.add("Ashish");
8.  v.add("Garima");
9.  Iterator<String> itr=v.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

```
Ayush
Amit
Ashish
Garima
```

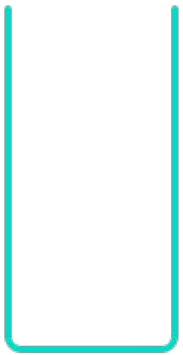
Stack

The stack is the subclass of Vector.

It implements the last-in-first-out data structure,

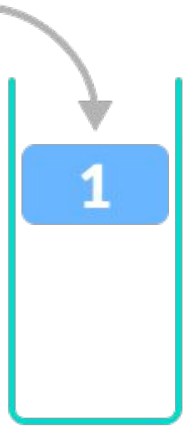
The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o),

TOP = -1



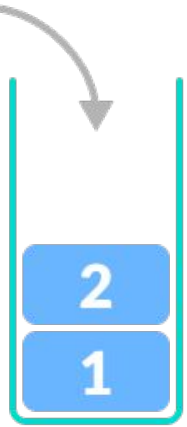
empty stack

TOP = 0
stack[0] = 1



push

TOP = 1
stack[1] = 2



push

TOP = 2
stack[2] = 3

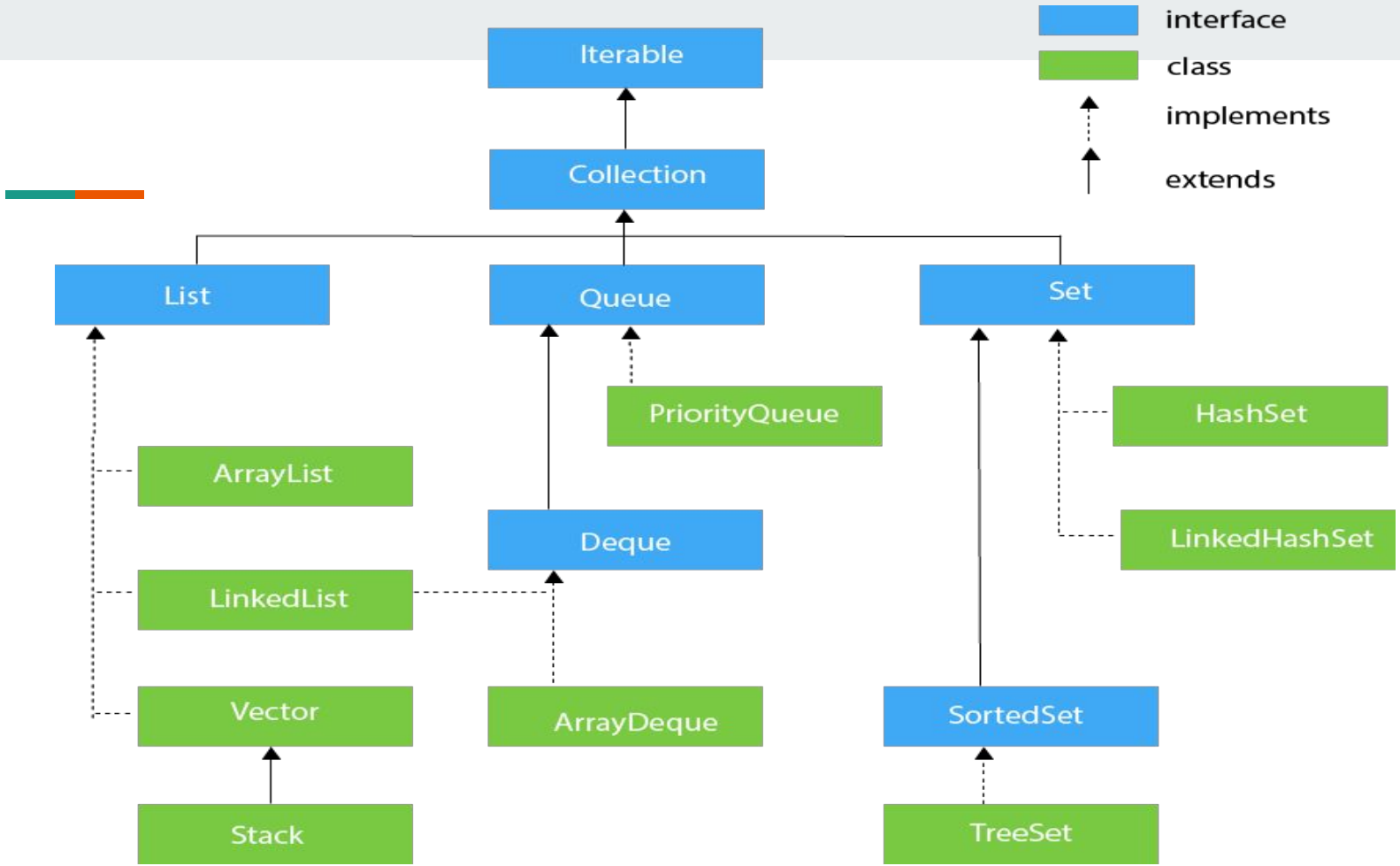
push

TOP = 1
return stack[2]

pop

```
1.  import java.util.*;
2.  public class TestJavaCollection4{
3.  public static void main(String args[]){
4.  Stack<String> stack = new Stack<String>();
5.  stack.push("Ayush");
6.  stack.push("Garvit");
7.  stack.push("Amit");
8.  stack.push("Ashish");
9.  stack.push("Garima");
10. stack.pop();
11. Iterator<String> itr=stack.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Ayush
Garvit
Amit
Ashish


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

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.

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

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.

it does not orders the elements in FIFO manner

```
1.  import java.util.*;
2.  public class TestJavaCollection5{
3.  public static void main(String args[]){
4.  PriorityQueue<String> queue=new PriorityQueue<String>();
5.  queue.add("Amit Sharma");
6.  queue.add("Vijay Raj");
7.  queue.add("JaiShankar");
8.  queue.add("Raj");
9.  System.out.println("head:"+queue.element());
10. System.out.println("head:"+queue.peek());
11. System.out.println("iterating the queue elements:");
12. Iterator itr=queue.iterator();
13. while(itr.hasNext()){
14. System.out.println(itr.next());
15. }
16. queue.remove();
17. queue.poll();
18. System.out.println("after removing two elements:");
19. Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
20. System.out.println(itr2.next());
}
}
```

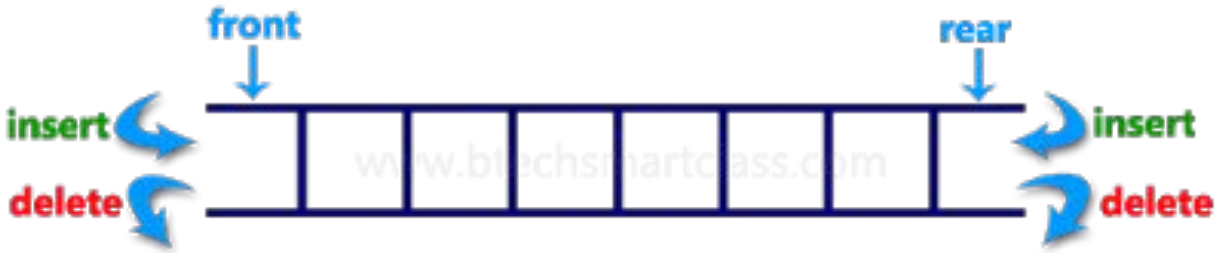
```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj
```

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.


front
rear
insert
delete
insert
delete

ArrayDeque

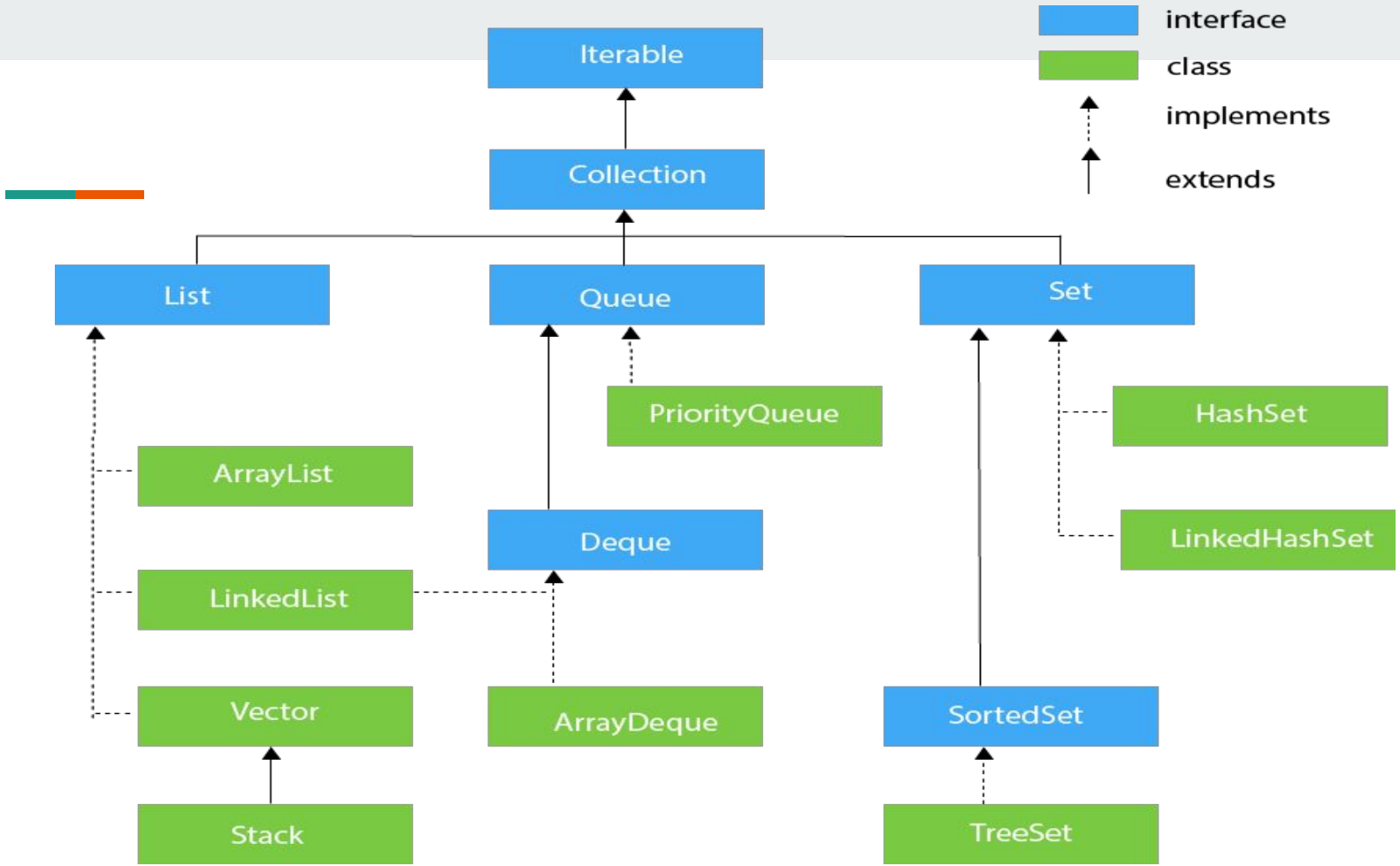
ArrayDeque class implements the Deque interface.

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.

```
1.  import java.util.*;
2.  public class TestJavaCollection6{
3.  public static void main(String[] args) {
4.  //Creating Deque and adding elements
5.  Deque<String> deque = new ArrayDeque<String>();
6.  deque.add("Gautam");
7.  deque.add("Karan");
8.  deque.add("Ajay");
9.  deque.poll();
10. //Traversing elements
11. for (String str : deque) {
12. System.out.println(str);
13. }
14. }
15. }
```

Karan
Ajay


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

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.

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>();

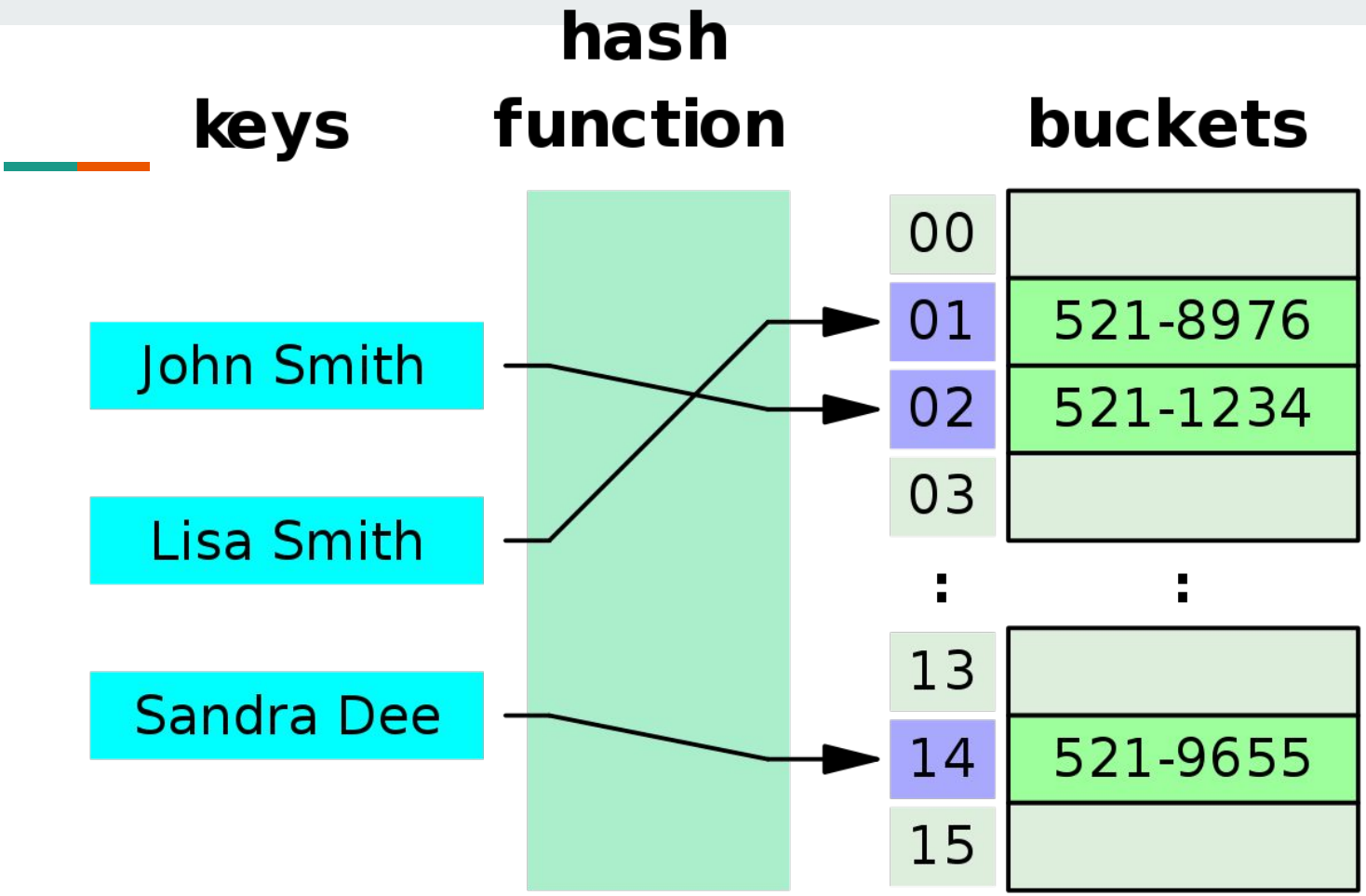
HashSet

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.


keys
hash function
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
13
14
15
521-8976
521-1234
521-9655

```
1.  import java.util.*;
2.  public class TestJavaCollection7{
3.  public static void main(String args[]){
4.  //Creating HashSet and adding elements
5.  HashSet<String> set=new HashSet<String>();
6.  set.add("Ravi");
7.  set.add("Vijay");
8.  set.add("Ravi");
9.  set.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

Vijay
Ravi
Ajay

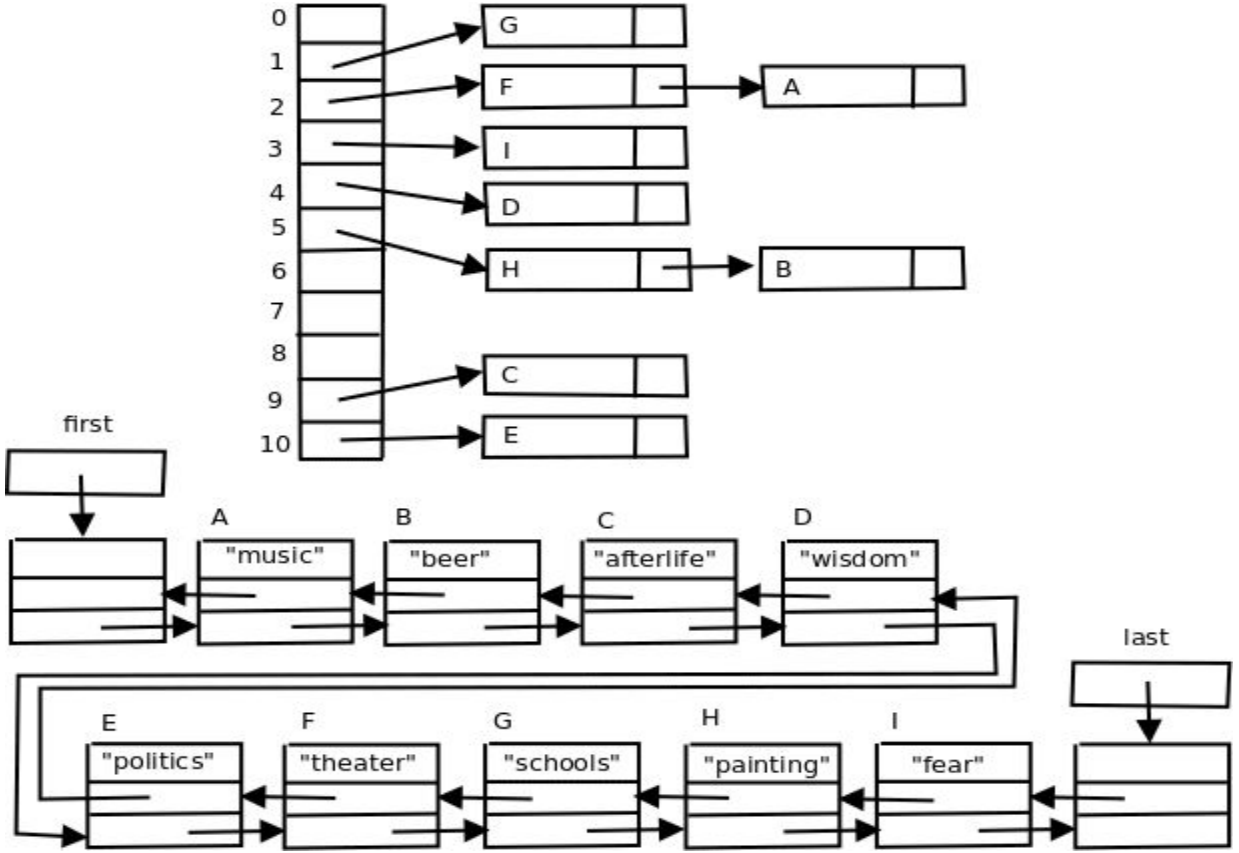
LinkedHashSet

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.


0
1
2
3
4
5
6
7
8
9
10
G
F
A
I
D
H
B
C
E
first
A
"music"
B
"beer"
C
"afterlife"
D
"wisdom"
last
E
"politics"
F
"theater"
G
"schools"
H
"painting"
I
"fear"

```
1.  import java.util.*;
2.  public class TestJavaCollection8{
3.  public static void main(String args[]){
4.  LinkedHashSet<String> set=new LinkedHashSet<String>();
5.  set.add("Ravi");
6.  set.add("Vijay");
7.  set.add("Ravi");
8.  set.add("Ajay");
9.  Iterator<String> itr=set.iterator();
10. while(itr.hasNext()){
11. System.out.println(itr.next());
12. }
13. }
14. }
```

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:

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

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage.

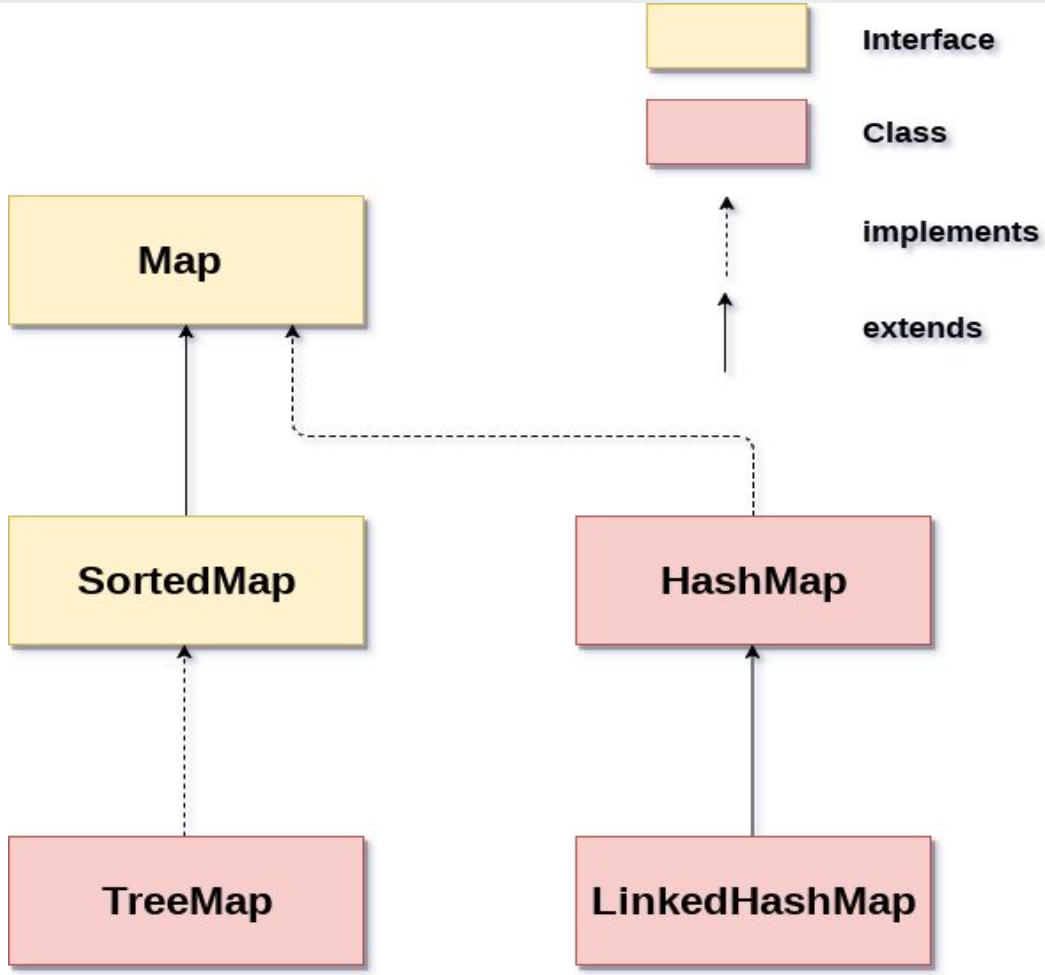
Like HashSet, TreeSet also contains unique elements.

However, the access and retrieval time of TreeSet is quite fast.

The elements in TreeSet stored in ascending order.

```
1.  import java.util.*;
2.  public class TestJavaCollection9{
3.  public static void main(String args[]){
4.  //Creating and adding elements
5.  TreeSet<String> set=new TreeSet<String>();
6.  set.add("Ravi");
7.  set.add("Vijay");
8.  set.add("Ravi");
9.  set.add("Ajay");
10. //traversing elements
11. Iterator<String> itr=set.iterator();
12. while(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }
```

```
Ajay
Ravi
Vijay
```


Interface
Class
implements
extends
Map
SortedMap
HashMap
TreeMap
LinkedHashMap

Java Map Interface

A map contains values on the basis of key, i.e. key and value pair.

Each key and value pair is known as an entry. A Map contains unique keys.

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

Class	Description
HashMap	HashMap is the implementation of Map, but it doesn't maintain any order.
LinkedHashMap	LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order.
TreeMap	TreeMap is the implementation of Map and SortedMap. It maintains ascending order.

Java HashMap

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

Java LinkedHashMap class

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

Java TreeMap class

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

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.

Comparable

```
1.  import java.util.*;
2.  import java.io.*;
3.  class Student implements Comparable<Student>{
4.  int rollno;
5.  String name;
6.  int age;
7.  Student(int rollno,String name,int age){
8.  this.rollno=rollno;
9.  this.name=name;
10. this.age=age;
11. }
```

```
1.  public int compareTo(Student st){
2.  if(age==st.age)
3.  return 0;
4.  else if(age>st.age)
5.  return 1;
6.  else
7.  return -1;
8.  }
9.  }
```

```
1.  public static void main(String
    args[]){
2.  ArrayList<Student> al=new
    ArrayList<Student>();
3.  al.add(new Student(101,"Vijay",23));
4.  al.add(new Student(106,"Ajay",27));
5.  al.add(new Student(105,"Jai",21));
6.
7.  Collections.sort(al);
8.  for(Student st:al){
9.  System.out.println(st.rollno+"
    "+st.name+" "+st.age);
10. }
11. }
12. }
```

Comparator

```
1.  import java.util.*;
2.  class AgeComparator implements Comparator<Student>{
3.  public int compare(Student s1,Student s2){
4.  if(s1.age==s2.age)
5.  return 0;
6.  else if(s1.age>s2.age)
7.  return 1;
8.  else
9.  return -1;
10. }
11. }
```

```
1.  public static void main(String args[]){
2.  //Creating a list of students
3.  ArrayList<Student> al=new ArrayList<Student>();
4.  al.add(new Student(101,"Vijay",23));
5.  al.add(new Student(106,"Ajay",27));
6.  al.add(new Student(105,"Jai",21));
7.
8.  System.out.println("sorting by Age");
9.  //Using AgeComparator to sort the elements
10. Collections.sort(al,new AgeComparator());
11. //Travering the list again
12. for(Student st: al){
13. System.out.println(st.rollno+" "+st.name+" "+st.age);
14. }
15.
16. }
```