using normal for loop to iterate the AL object:

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

        Student student= students.get(i);

        System.out.println("Roll is :"+student.getRoll());
        System.out.println("Name is :"+student.getName());
        System.out.println("Marks is :"+student.getMarks());

        System.out.println("================================");

}
```

Note : from the List object we can get elements one by one by using following approaches also:

1.by using Iterator obj
2.by using ListIterator obj
3.by using forEach() method
4.by using Stream api

--in addition to the normal and enhanced for loop.

where as from Set and Queue:

--we can not use Normal for loop we can only use :

1.enhanced for loop
2.by using Iterator
3.by using forEach() method
4.by using Stream api

LinkedList:
========

--it is also one of the implementation of List interface.

--LinkedList class from Java 1.5 onwards also implements Deque interface.

--LinkedList class is the best choice if our frequent opeation is insertion or deletion from the middle.

--LinkedList class also follows the properties of List and Deque(preserve the sequence and index concept)

10 20 30 50 60
0 1 2 3 4 5

With the LinkedList if we delete or insert elements then too much siffting operation is not required.

--In Java LinkedList is implemented using Doubly linked list data structure.

example:

```
package com.masai;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Scanner;

public class Demo {

	public static void main(String[] args) {


		LinkedList<Integer> ll = new LinkedList<>();

		ll.add(10);
		ll.add(20);
		ll.add(30);
		ll.add(40);
		ll.add(50);
		ll.add(null);
		ll.add(null);
		ll.add(60);
		ll.add(60);

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

```
//                  for(int x:ll) {  // here the chance of NPE
//                          System.out.println(x);
//                  }

                    for(Integer x:ll) {
                            System.out.println(x);

                    }

            }
}
```

Vector:
=======

--it is also similar to ArrayList class, with following differences.

1.AL introduced in java 1.2 v whereas Vector class is a legecy collection class introduced in java1.0 v

2.where AL reaches to the max capacity the new AL obj will be created internally with newCapacity = (currentCapacity * 3/2)+1
where as when Vector class reaches to its max capacity then a new vector obj is created in the memory by
double capacity.

3. most of the methods of AL class is non-synchronized (not thread safe) where as most of the methods of Vector class is synchronized, i.e thread-safe.

---AL will give better and fast performance compare to Vector class.

totalticket = 10;

8

9

```
public synchronized void bookTicket(int numberOfTicket){
//
if(numberOfTicket < = totalTicket)
{
totalTicket -= numberOfTicket;
}
}

public synchronized void viewAvailability(){


}
```

--until we have specific requierement we should not use synchoronized keyword.

---we have an option to make our AL objects methods as synchronized.

java.util.Collection(I) : root interface of Collection f/w
java.util.Collections(c) : utility class.

Collections.sychronizedList(al); // it will convert the AL to the synchronized List(thread safe obj)

Vector --- black & white tv

AL ---> Color TV -----> reduce the color

Stack class:
==========

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

                st.add(10);
                st.add(20);
                st.add(30);
                st.add(40);
                st.add(50);
                st.add(null);
                st.add(null);
                st.add(60);
                st.add(60);
```

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

                System.out.println(st.pop());

                System.out.println(st);

        }

}
```

--All the opeation of the Stack we can perform with the help of LinkedList class also.

```
ArrayList<String> al = new ArrayList<>();  /// too specific

List<String> al= new ArrayList<>(); // recommended way.

Collection<String> al = new ArrayList<>();//

Object al = new ArrayList<>(); // too generic
```

example

Demo.java:

```
package com.masai;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import java.util.Stack;
import java.util.Vector;

public class Demo {

                public List<Student> getStudents(){
```

```
                LinkedList<Student> students = new LinkedList<>();

                students.add(new Student(10, "n1", 780));
                students.add(new Student(12, "n2", 580));
                students.add(new Student(13, "n3", 780));
                students.add(new Student(14, "n4", 680));


                return students;

        }



    public static void main(String[] args) {


        Demo d1= new Demo();

        List<Student> students= d1.getStudents();


        for(Student student:students) {
            System.out.println(student);
        }


    }

}
```

Set:
====

--it is the child interface of the Collection interface.

--it defines the behaviour of a collection that it does not allows the duplicate elements.

--here index concept is not applicable. so we can not use get(int index) method.

HashSet(c):
=========

--it is the first implementation of the Set interface.

--here insertion order is not preserved,

--elements will be added based on their hashCode.

--duplicate elements are not allowed.

--if we try to add any duplicate element, it does not throw any exception, simply add() method return false.

--null insertion is possible , but only one null value.

******HashSet class is the best choice, if our frequent operation is searching.**

--searching a paerticular element based on hashCode will have time complexity O(1).

HashSet<Integer> hs= new HashSet<>();

--here an empty HS obj is created with the initial capacity 16 and the default load factor(fill ratio) is 0.75

--here fill ratio means after completion of 75% the new HS object will be created in the memory.

HashSet<Integer> hs= new HashSet<>(1000,0.8f);

--here initial capacity is 1000 and and once reaches to the 80% then a new HS object is created in the memory.

example:

```
HashSet<Integer> hs= new HashSet<>();

    hs.add(10);
    hs.add(20);
    hs.add(30);
```

```
hs.add(10);
hs.add(10);
hs.add(null);
hs.add(null);



System.out.println(hs.size());
System.out.println(hs);
```

output:

```
4
[null, 20, 10, 30]
```

Note: to access the elements one by one from the HS class we can not normal for loop, but we can use enahanced forloop

example

```
for(Integer i1:hs) {
	System.out.println(i1);
}
```

Note: HashSet is very much related with HashMap, it internally uses the HashMap to store the element.

Object equality:
==========

--equals() method belongs to Object class,

```
public boolean equals(Object obj);
```

--this method is implemented inside the Object class as follows:

```
public boolean equals(Object obj){
/*
if(obj == this)
return true;
```

```
else
return false;
*/

return obj == this;



}
```

--in order to make our class objects logically equal we need to override the above equals() method from the Object class to our Student class.

ex:

Inside Student.java:

```
	@Override
	public boolean equals(Object obj) {

		Student s1= this;
		Student s2=  (Student)obj;

//if(s1.getRoll() == s2.getRoll()  && s1.getName().equals(s2.getName())  && s1.getMarks() == s2.getMarks() )
//			return true;
//		else
//			return false;


return (s1.getRoll() == s2.getRoll()  && s1.getName().equals(s2.getName())  && s1.getMarks() == s2.getMarks() );







	}
```

Note: this equals() method has a best friend called hashCode() method, it is also defiend inside the Object class:

public int hashCode();

---equals() and hashCode method is like a contract, if we override the equals() method to make our objects logically equal then we have to override the hashCode() method also;

--if we call equals() method to compare two object and if it returns true then those objects hashCode value should also be same.

Student.java:
==========

```
package com.masai;

import java.util.Objects;

public class Student {

	private int roll;

	private String name;

	private int marks;


	public Student() {
		// TODO Auto-generated constructor stub
	}


	@Override
	public boolean equals(Object obj) {

		Student s1= this;
		Student s2=  (Student)obj;

//		if(s1.getRoll() == s2.getRoll()  && s1.getName().equals(s2.getName())  && s1.getMarks() == s2.getMarks() )
//			return true;
//		else
```

```
//                      return false;


                return (s1.getRoll() == s2.getRoll()  && s1.getName().equals(s2.getName())
&& s1.getMarks() == s2.getMarks() );


        }


        @Override
        public int hashCode() {

                return Objects.hash(roll,name,marks);

        }


        public Student(int roll, String name, int marks) {
                super();
                this.roll = roll;
                this.name = name;
                this.marks = marks;
        }


        public int getRoll() {
                return roll;
        }


        public void setRoll(int roll) {
                this.roll = roll;
        }


        public String getName() {
                return name;
        }


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

```
	public int getMarks() {
		return marks;
	}


	public void setMarks(int marks) {
		this.marks = marks;
	}


	@Override
	public String toString() {
		return "Student [roll=" + roll + ", name=" + name + ", marks=" + marks + "]";
	}



}
```

Demo.java:
========

```
package com.masai;

import java.util.HashSet;

public class Demo {

	public static void main(String[] args) {




	HashSet<Student> hs=new HashSet<>();

	hs.add(new Student(10, "n1", 780));
	hs.add(new Student(12, "n2", 880));
	hs.add(new Student(13, "n3", 980));
	hs.add(new Student(10, "n1", 780));
```

```
        hs.add(new Student(10, "n1", 780));

        System.out.println(hs.size());


        }
}
```