Java

Wrapper Classes

Object Class

Object class defined in java.lang package is the superclass of all other classes defined in Java programming language. Every class extends from the Object class either directly or indirectly.

Methods define in Object class.

clone() - Creates and returns a copy of this object.

equals() - Indicates whether some other object is "equal to" this one.

finalize() - Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

getClass() - Returns the runtime class of an object.

hashCode() - Returns a hash code value for the object.

notify() - Wakes up a single thread that is waiting on this object's monitor.

notifyAll() - Wakes up all threads that are waiting on this object's monitor.

toString() - Returns a string representation of the object.

wait() - Causes current thread to wait until another thread invokes the notify() or notifyAll() method for this object.

equals() and ***hashcode()***.

equals() and *hashcode() are one of the two method of cosmic class java.lang.Object*

Used to compare the objects.

equals() :

Signature : public boolean equals() { }

The default implementation provided by the JDK is based on memory location — two objects are equal if and only if they are stored or referred in the same memory address (override equals() method to check the equality as per your requirement.)

hashcode () :

Signature : public boolean equals() { }

*T*his method returns a random integer that is unique for each instance. This integer might change between several executions of the application and won't stay the same.

Object class Example

Example 1

```
package com.corejava.collections.learning;

public class ObjectTest {

	public static void main(String[] args) {
		Student1 obj1=new Student1(1,"Ramesh");
		Student1 obj2=new Student1(2,"Suresh");
		Student1 obj3=new Student1(1,"Ramesh");

		/*
* If we don't override equals() method in Student1 class then the default equals method from the
Object class check the reference of the object e.g return (obj1 == obj2);
Try commenting equals method. If do so then the below equality will return false as both bj1 and
obj2 have different reference (as they are created by new)
		*/
		System.out.println("obj1.equals(obj3) : "+obj1.equals(obj3));

		System.out.println("Obj1 hashCode:"+obj1.hashCode());
		System.out.println("Obj2 hashCode:"+obj2.hashCode());
		System.out.println("Obj2 hashCode:"+obj3.hashCode()+"\n");

		Student1 s1=new Student1(5,"Kedar");
		Student1 s2=s1;
		System.out.println("Hash Code of S1:"+s1.hashCode());
		System.out.println("Hash Code of S1:"+s2.hashCode());
	}
}

class Student1 {
	int id;
	String name;

	public Student1(int id, String name) {
	  this.name = name;
	  this.id = id;
    }

	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public boolean equals(Object s1) {
```

```
		System.out.println("In equals method of Student class");
		if(s1==null)
			return false;
		if(!(s1 instanceof Student1))
			return false;
		Student1 s = (Student1)s1;
		if(this.id==s.getId() && this.name==s.getName())
			return true;
		else
			return false;
	}
	/* If we don't implement hash code then even if Objects are equal in nature
	 * their hashcode will not be same(not satisfying the contract)
	 */
	public int hashCode() {
		return this.id * 25 + this.name.length();
	}
	/* Remember
		1) If two objects are equal, then they must have the same hash code.
		2) If two objects have the same hash code, they may or may not be equal.

	*/
}
```

Example2:
Contract of equals() and hashCode()

If two objects are equal according to the *equals(Object)* method, then calling the *hashcode()* method on each of the two objects must produce the same integer result.

```
public class Student {
   private int id;
   private String name;

   public Student(int id, String name) {
        this.name = name;
         this.id = id;
   }
   public int getId() {  return id;   }
   public void setId(int id) {       this.id = id;    }

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

public class HashcodeEqualsTest {
   public static void main(String[] args) {
      Student s1 = new Student(1,"Kunal");
      Student s2 = new Student(1,"Kunal");
```

```
        System.out.println("S1 hashcode =  "+ s1.hashCode());
        System.out.println("S2hashcode = "+ s2.hashCode());
        System.out.println("Equality between s1 and s2= "+ s1.equals(s2));
    }
}
/*
 Output
 S1 hashcode = 1993134103
 S2 hashcode = 604107971
 Equality between s1 and s2= false

 comment : If you want to get the equals and hashCode() contract true then we need to implement equals and
hascode
 method.
 */
```

Interface

An interface in Java is a mechanism that is used to achieve complete abstraction. It is basically a kind of class that contains only constants and abstract methods.

- Data members declared in an interface are by default public, static, and final.
- Only abstract and public modifiers are allowed for methods in interfaces
- We cannot create an object of interface using new operator. But we can create a reference of interface type and interface reference refers to objects of its implementation classes
- From Java 8 onwards, we can define static and default methods in an interface. Prior to Java 8, it was not allowed.
- A variable in an interface must be initialized at the time of declaration.

Comparator interface

The Java Comparator interface, java.util.Comparator, represents a component that can compare two objects so they can be sorted using sorting functionality.

The Java Comparator interface definition looks like this:

```
public interface Comparator<T> {

    public int compare(T o1, T o2);

}
```

Notice that the Java Comparator interface only has a single method. This method, the compare() method, takes two objects which the Comparator implementation is intended to compare. The compare() method returns an int which signals which of the two objects was larger. The semantics of the return values are:

A negative value means that the first object was smaller than second object.

The value 0 means the two objects are equal.

A positive value means that the first object was larger than the second object.

You will find the Example of comparator below in Arraylist

Collection

Collection (I) : Group of element will be represented by single entity.

It is root interface for all the Collection.

All classes and interfaces are part of java.util package.

Common methods applicable for any Collection object:

boolean add(Object o)

boolean isEmpty()

boolean addAll(Collection c)

boolean contains(Object o)

boolean remove()

boolean containsAll(Collection c)

void clear()

Object[] toArray()

boolean retainAll(Collection c)

Iterator iterator()

boolean removeAll(Collection c)

int size()

List interface

In Java, the **List** interface is an ordered collection that allows us to store and access elements sequentially. It extends the Collection interface.

Lists typically allow duplicate elements

List is an interface; we cannot create objects from it.

Features of the List

1. The list allows storing duplicate elements in Java. JVM differentiates duplicate elements by using 'index' (Position). It always starts at zero.

2. In the list, we can add an element at any position.

3. It maintains insertion order. i.e., List can preserve the insertion order by using the index.

4. It allows for storing many null elements.

5. Java list uses a resizable array for its implementation. Resizable means size can grow, or we can increase or decrease the size of the array.

6. Except for LinkedList, ArrayList, and Vector is an indexed-based structure.

7. It provides a special Iterator called a ListIterator that allows accessing the elements in the forward direction using hasNext() and next() methods.

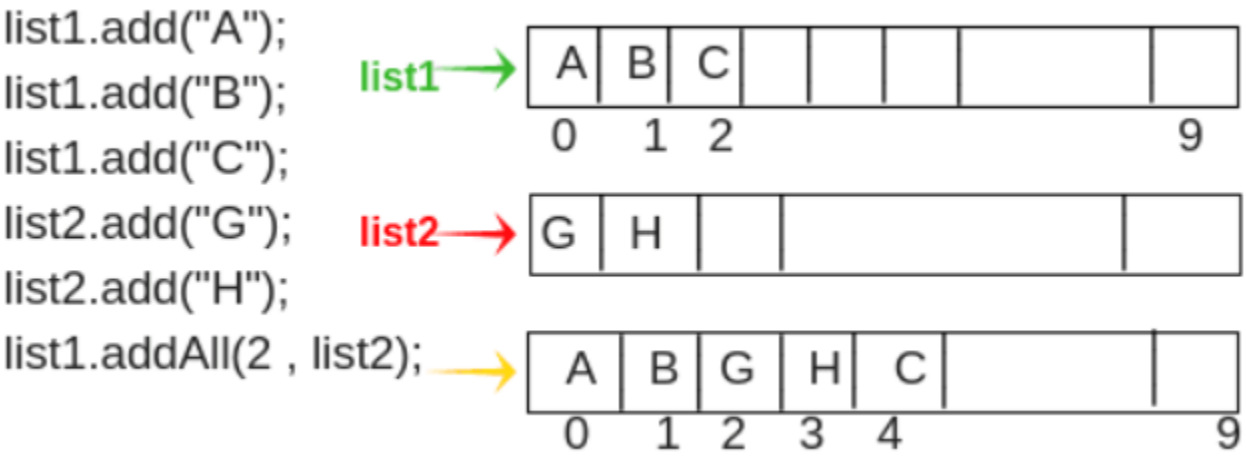
In order to use functionalities of the List interface, we can use these classes:

- ArrayList
- LinkedList
- Vector
- Stack

ArrayList

The `ArrayList` class is a resizable array, which can be found in the `java.util` package.

The difference between a built-in array and an `ArrayList` in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an `ArrayList` whenever you want.



Where does ArrayList store its element

Internally it uses array to store its element. It's an Object array which is defined as follows.

transient Object[] elementData;

Constructor

Initial capacity of the created ArrayList depends on the constructor used.

- ArrayList(int initialCapacity)– If initial capacity is explicitly specified while constructing an ArrayList then it is ensured that the elementData array is created with that length.

  Internal java Code:

  this.elementData = new Object[initialCapacity];

- ArrayList()– If no initial capacity is specified then the ArrayList is created with the default capacity 10

Internal java Code:

```
private static final int DEFAULT_CAPACITY = 10;
public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

Basic ArrayList Example

```
package com.corejava.itp.collection;

import java.util.ArrayList;
import java.util.List;
import java.util.Vector;

/*
* Resizable-array implementation of the List interface.
 */

public class ArrayListBasics {
        public static void main(String[] args) {
                // Integer a=10; // primitive to wrapper.
                ArrayList myList = new ArrayList();  // default capacity(length or size) is 10
                myList.add(10);  // int 10 will be converted to Integer wrapper class.
                myList.add(20);
                myList.add(30);
                myList.add(40);
                myList.add(50);
                myList.add(60);
```

```
myList.add(70);
myList.add(80);
myList.add(90);
myList.add(100); // size/capacity will be increase
myList.add(110);
myList.add("ONE");
myList.add("Two");
myList.add("Three");

System.out.println(myList);
System.out.println("ArrayList size :"+myList.size());
System.out.println("Element at index 0:"+myList.get(0));      // array[0]
System.out.println("Element at index 2:"+myList.get(2));      // array[2]
System.out.println("Element at index 10:"+myList.get(10));      // array[10]
//System.out.println("Element at index 50:"+myList.get(50));      // array[10]

System.out.println("Does ArrayList contains 30??: "+myList.contains(30));
System.out.println("Does ArrayList contains 300??: "+myList.contains(300));
System.out.println("Remove element at index 5:"+myList.remove(5));
System.out.println("ArrayList after remove :"+myList);

System.out.println("ArrayList travel : ");
for(int i=0;i<myList.size();i++) {
        System.out.println(myList.get(i));
}
```

// Define the ArrayList with specific type of element. E.g. If ArrayList will have only int.

```
// Below ArrayList will hold only integer.
ArrayList<Integer> intList = new ArrayList<Integer>();
```

```
intList.add(101);
intList.add(102);
intList.add(103);
intList.add(104);

//print ArrayList using for each
System.out.println("\nInteger arraylist printed using for-each loop");
for(Integer i:intList) {
        System.out.println(i);
}
ArrayList<Integer> studRoll=new ArrayList<Integer>();
studRoll.addAll(intList);
studRoll.add(105);
studRoll.add(106);
System.out.println("Stud Roll Numbers :"+studRoll);

// ArrayList Constructor
ArrayList<String> strList = new ArrayList<String>(); // Initial size will be 10
ArrayList<String> strList1 = new ArrayList<String>(100); // Initial size/capacity will be 100

//Another way to define the ArrayList using interface reference.
// Imp: We can't create object of an interface
//List aList=new List();     //not valid- should not create interface object.
List aList= new ArrayList<Integer>();

/*
 * Difference between below statements
 ArrayList<Integer> intList = new ArrayList<Integer>();
 List aList= new ArrayList<Integer>();
```

```
                intList can't be converted to other list implementation type.
                aList can be converted to other List implementation type like  ArrayList, Vector and LinkedList
               */
               aList = new Vector<Integer>();
               // intList = new Vector<Integer>(); //error in this list as intList cant convert to Vector.

        }
}
```

Advance Example of ArrayList

```
package com.corejava.collections.learning;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;

public class ArrayListDemo {
  public static void main(String[] args) {

        ArrayList aList = new ArrayList();
        //aList = new Vector(); you can't covert it to Vector as aList is declared as ArrayList.

        // Below declaration is more suitable, Program to interface and not to implementation
        // i.e create a reference of supertype and assign memory of subtype.
        List aList1 = new ArrayList();
```

```
// ArrayList with only integer object. Initial capacity is 10
ArrayList<Integer> intList = new ArrayList<Integer>();

// ArrayList with String object with initial capacity 20
ArrayList<String> strList = new ArrayList<String>(20);

aList.add("Zero");
aList.add("One");
aList.add("two");
aList.add(3);
aList.add("Four");

System.out.println("0th Element of aList using get(index):"+aList.get(0));
System.out.println("Complete array List :"+aList);
regularforWay(aList);
forEachWay(aList);
iteratorWay(aList);

System.out.println("\nArrayList contains method, check for Four: "+aList.contains("Four"));
System.out.println("\nArrayList remove with index method, remove 3: "+aList.remove(3));
System.out.println("\nArrayList set (replace) with index method: "+aList.set(2,"twoo"));

strList.add("Five");
strList.add("Seven");
strList.add("One");
strList.add("Four");
```

```
        strList.add("Nine");

        sortList(strList);
        sortListInReverse(strList);
        customSort(strList);
        ArrayList<String> revList = reverseArrayList(strList);

}

        static void regularforWay(List a) {
                int i=0;
                System.out.println("\nArraylist using regular for loop");
                for(i=0;i<a.size();i++) {
                        System.out.println(a.get(i));
                }
        }

        static void forEachWay(List a) {
                System.out.println("\nArraylist using foreach loop\n");
                for(Object s:a) {
                                System.out.println(s);
                }
        }
        static void iteratorWay(List a) {
                System.out.println("\nArraylist using iterator\n");
                Iterator it=a.iterator();
                while(it.hasNext()) {
```

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

        static void sortList(List<String> aList) {
            System.out.println("\nSort using Collections");
            System.out.println("Before :"+aList);
            Collections.sort(aList);
            System.out.println("After :"+aList);
        }

        static void sortListInReverse(List<String> aList) {
            System.out.println("\nReverse Sort using Collections");
            System.out.println("Before :"+aList);
            Collections.sort(aList,Collections.reverseOrder());
            System.out.println("After :"+aList);
        }

        static void customSort(List<String> aList) {
            // Implement comparator interface
            System.out.println("\nSort using Comparator");
            System.out.println("Before:"+aList);
            Collections.sort(aList,new MySortComparator());
            System.out.println("After:"+aList);
        }

    static ArrayList<String> reverseArrayList(ArrayList<String> alist)  {
```

```
        // ArrayList for storing reversed elements
        ArrayList<String> revArrayList = new ArrayList<String>();
        for (int i = alist.size() - 1; i >= 0; i--) {
            // Append the elements in reverse order
            revArrayList.add(alist.get(i));
        }
        // Return the reversed arraylist
        return revArrayList;
    }
}
class MySortComparator implements Comparator<String>{
        @Override
        public int compare(String o1, String o2) {
                // o1>o1 it will return positive
                // o1==o2 it will return zero
                // o1<o2 return negative
                return o1.compareTo(o2);
        }
}
```

Another Example of ArrayList with Comparator implementation.

```
package com.corejava.collections.learning;
import java.lang.*;
import java.util.*;

class Student {
    int rollno;
```

```
    String name, address;

    // Constructor
    public Student(int rollno, String name, String address)   {
        this.rollno = rollno;
       this.name = name;
        this.address = address;
    }

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name + " "  + this.address;
    }
}

class Sortbyroll implements Comparator<Student> {
    // Used for sorting in ascending order of roll number
    public int compare(Student a, Student b)  {
        return a.rollno - b.rollno;
    }
}

class Sortbyname implements Comparator<Student> {
    // Used for sorting in ascending order of name
    public int compare(Student a, Student b)  {
        return a.name.compareTo(b.name);
```

```
    }
}

// Main class
public class ArrayListCustomSort {
    public static void main(String[] args)  {
        ArrayList<Student> ar = new ArrayList<Student>();
        ar.add(new Student(601, "Ram", "pune"));
        ar.add(new Student(571, "Kunal", "delhi"));
        ar.add(new Student(491, "Samar", "jaipur"));
        ar.add(new Student(491, "Samir", "jaipur"));

        System.out.println("Unsorted");
        for (int i = 0; i < ar.size(); i++)
            System.out.println(ar.get(i));

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

        System.out.println("\nSorted by rollno");
        for (int i = 0; i < ar.size(); i++)
            System.out.println(ar.get(i));

        Collections.sort(ar, new Sortbyname());
        System.out.println("\nSorted by name");
        for (int i = 0; i < ar.size(); i++)
            System.out.println(ar.get(i));
    }
```

```
}
/* Output
Unsorted
601 Ram pune
571 Kunal delhi
491 Samar jaipur
491 Samir jaipur

Sorted by rollno
491 Samar jaipur
491 Samir jaipur
571 Kunal delhi
601 Ram pune

Sorted by name
571 Kunal delhi
601 Ram pune
491 Samar jaipur
491 Samir Jaipur   */
```

Interview question –

1) reverse the list without using temp list

2) How does add/remove method work in ArrayList

   Answer:

   ArrayList is internally implemented as growable or resizable Array.

If you see the ArrayList internal implementation in Java, everytime add() method is called it is ensured that ArrayList has required capacity.

If the capacity is exhausted a new array is created with 50% more capacity than the previous one. All the elements are also copied from the previous array to the new array.

How does remove method work in ArrayList

If you remove any element from an array then all the subsequent elements are to be shifted to fill the gap created by the removed element.

LinkedList

Java LinkedList class uses a doubly linked list to store the elements.

Features

- 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.

LinkedList constructor and implementation

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.

//Definition of LinkedList class

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

{

}

```
package com.corejava.collections.learning;

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

  LinkedList<String> langList=new LinkedList<String>();
```

```
  langList.add("Java");
  langList.add("CPP");
  langList.add("Python");
  langList.add("R-lang");

  System.out.println("Print the newly created LinkedList:");

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

  // Adding an element at the specific position
  langList.add(1, "React");
  System.out.println("After add(int index, E element) method: "+langList);

  // Adding an element at the first position
  langList.addFirst("Angular");
  System.out.println("After invoking addFirst(E e(i.e 'Angular')) method:
"+langList);
  //Adding an element at the last position
  langList.addLast("AWS-Cloud");
  System.out.println("After invoking addLast(E e(i.e 'AWS-Cloud')) method:
"+langList);

  //Removing specific element from arraylist
  langList.remove("CPP");
  System.out.println("After invoking remove(object) method-removed 'CPP':
"+langList);
  //Removing element on the basis of specific position
  langList.remove(0);
  System.out.println("After invoking remove(index) method- removed at index
0: "+langList);
  langList.removeFirst();
  System.out.println("After invoking removeFirst() method"+langList);
  langList.removeLast();
  System.out.println("After invoking removeLast() method"+langList);

  langList.add("Java-Spring");
  langList.add("Java-Spring");
  langList.add("Java-Spring");
  langList.add("Java-Spring");
```

```
  System.out.println("After invoking add('Java_Spring') method 4
times"+langList);

  langList.remove("Java-Spring");
  System.out.println("After invoking remove('Java_Spring')
method:"+langList);

  langList.removeFirstOccurrence("Java-Spring");
  System.out.println("After invoking removeFirstOcurrence()
method"+langList);
  langList.removeLastOccurrence("Java-Spring");
  System.out.println("After invoking removeLastOcurrence()
method"+langList);

  System.out.println("Print the LinkedList in reverse Order");
  reverseLinkedList(langList);
 }

 static void reverseLinkedList(LinkedList langList) {
      Iterator itr =langList.descendingIterator();
      while(itr.hasNext()){
        System.out.println(itr.next());
      }
 }
}
```

Apart from regular List methods LinkedList provides few extra methods as below.

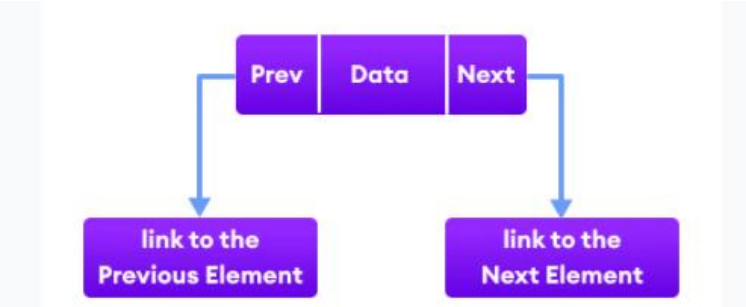
Methods	Descriptions
addFirst()	adds the specified element at the beginning of the linked list
addLast()	adds the specified element at the end of the linked list
getFirst()	returns the first element
getLast()	returns the last element

removeFirst()	removes the first element
removeLast()	removes the last element
peek()	returns the first element (head) of the linked list
poll()	returns and removes the first element from the linked list
offer()	adds the specified element at the end of the linked list

How does LinkedList class store its element?

Internally LinkedList class in Java uses objects of type **Node** to store the added elements. Node is implemented as a static class with in the LinkedList class.

```
private static class Node<E> {
  E data;
  Node<E> next;
  Node<E> prev;
  Node(Node<E> prev, E element, Node<E> next) {
    this.item = element;
    this.next = next;
    this.prev = prev;
  }
}
```



How add() method works in a LinkedList?

Since it is a linked list so apart from regular **add()** method to add sequentially there are **addFirst()** and **addLast()** methods also in Java LinkedList class.

There are also separate variables for holding the reference of the first and last nodes of the linked list.

```
/**
* Pointer to first node.
*/
transient Node<E> first;
/**
* Pointer to last node.
*/
transient Node<E> last;
```

Add() in LinkedList

```
private void linkFirst(E paramE) {
   Node<E> node1 = this.first;
   Node<E> node2 = new Node<>(null, paramE, node1);
   this.first = node2;
   if (node1 == null) {
     this.last = node2;
   } else {
     node1.prev = node2;
   }
   this.size++;
   this.modCount++;
 }
```

Add() element at last position

```
/**
 * Links e as last element. Called when add() or addLast() is invoked in a program.
 */
void linkLast(E e) {
  final Node<E> l = last;
  final Node<E> newNode = new Node<>(l, e, null);
  last = newNode;
  if (l == null)
    first = newNode;
  else
    l.next = newNode;
  size++;
  modCount++;
}
```

ArrayList and LinkedList Difference

1) Search/Retrieve

ArrayList search is faster than LinkedList search as ArrayList implement RandomAccess Interface.

get(int index) in ArrayList gives the performance of O(1)

get(int index) in LinkedList performance is O(n).

2) Insert

LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case.

3) Deletion

LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element)

4) Memory Overhead

ArrayList maintains indexes & element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

5) Underline Implementation

ArrayList : Underline data structure is Resizable and growable Array

LinkedList: Underline data structure is doubly linked list.

6) Parent interface.

ArrayList implements RandomAccess interface.

LinkedList does not implement RandomAcess interface.

When to use ArrayList or LinkedList

If there is a requirement of frequent addition and deletion in an application, then LinkedList is a best choice.

Search (get method) operations are fast in Arraylist (O(1)) but not in LinkedList (O(n)) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best choice.

Vector

The Vector class is an implementation of the List interface that allows us to create resizable-arrays similar to the ArrayList class.

The Vector class synchronizes each individual operation. This means whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation.

Simple example

```
import java.util.Vector;

class Main {
    public static void main(String[] args) {
        Vector<String> mammals= new Vector<>();

        // Using the add() method
        mammals.add("Dog");
        mammals.add("Horse");

        // Using index number
        mammals.add(2, "Cat");
        System.out.println("Vector: " + mammals);

        // Using addAll()
        Vector<String> animals = new Vector<>();
        animals.add("Crocodile");

        animals.addAll(mammals);
        System.out.println("New Vector: " + animals);
    }
}
```

Another example

package com.corejava.collections.learning;

import java.util.Collections;

```
import java.util.Enumeration;
import java.util.Iterator;
import java.util.Vector;

public class VectorDemo {
  public static void main(String[] args) {
          Vector<String> langVector=new Vector<String>();
          langVector.add("Java");
          langVector.add("CPP");
          langVector.add("Python");
          langVector.add("R-lang");

          System.out.println("Print the complete vector object: "+langVector);
          System.out.println("Capacity of the vector:"+langVector.capacity());
          System.out.println("Does vector has CPP :"+langVector.contains("CPP"));
          System.out.println("get the element at index 3:"+langVector.get(3));
          System.out.println("get the index of 'Python'"+langVector.indexOf("Python"));
          System.out.println("set method to replace java->JAVA"+langVector.set(0, "JAVA"));

          // In iterator you can travel the collection as well as remove the elements from it.
          // Using Enumeration You can't do the modification(add/remove) inside collection.
          // getting the Enumeration object over Vector
         Enumeration enumeration = Collections.enumeration(langVector);
        System.out.println("printing each enumeration constant by enumerating through the Vector:");
        while (enumeration.hasMoreElements()) {
          System.out.println(enumeration.nextElement());
       }
   }
}
```

Stack

The Java collections framework has a class named `Stack` that provides the functionality of the stack data structure.

The `Stack` class extends the `Vector` class.

In order to create a stack, we must import the java.util.Stack package first. Once we import the package, here is how we can create a stack in Java.

Stack<Type> stacks = new Stack<>();

Here, Type indicates the stack's type. For example,

// Create Integer type stack

Stack<Integer> stacks = new Stack<>();

// Create String type stack

Stack<String> stacks = new Stack<>();

Sample Code

```
import java.util.Stack;

class Main {
   public static void main(String[] args) {
      Stack<String> animals= new Stack<>();

      // Add elements to Stack
      animals.push("Dog");
      animals.push("Horse");
```

```
        animals.push("Cat");
        System.out.println("Initial Stack: " + animals);

        // Remove element stacks
        String element = animals.pop();
        System.out.println("Removed Element: " + element);
    }
    // Access element from the top using peek() method.
        String element = animals.peek();
        System.out.println("Element at top: " + element);
    // Check if stack is empty
        boolean result = animals.empty();
        System.out.println("Is the stack empty? " + result);
}
```

Short summary of List

```
/*
ArrayList vector and LinkedList difference

Internal logic
ArrayList : Array
LinkedList : doubly Linked list
Vector : Array

Declaration
ArrayList ->   ArrayList a=new ArrayList()  or List a=new
ArrayList();
LinkedList ->   LinkedList list=new LinkedList()  or List
list=new LinkedList();
Vector ->   Vector v=new Vector()  or List v=new Vector();
```

```
Synchronized - main difference
ArrayList : Methods are non Synchronized
LinkedList : Methods are non Synchronized
Vector : Methods are Synchronized means Vectors are thread
safe.

When to use what
ArrayList : When retrieve operation is more. Because it has
Random access a[5] or a[1000]
LinkedList : Retrieve operations are slow. Use when Insert or
remove is frequent operation.
Vector : Use Vector then program is multi-threaded program.
When retrieve operation is more. Because it has Random access
a[5] or a[1000]

*/
```

Iterators in Java

Iterators in Java are used to retrieve the elements one by one from a collection object. They are also called cursors in java

There are four types of iterators or cursors available in Java. They are as follows:

- Enumeration
- Iterator
- ListIterator
- Spilterator (Java 1.8)

Enumeration in Java

Enumeration is the first iterator that was introduced in Java 1.0 version. It is located in java.util package. It is a legacy interface that is implemented to get elements one by one from the legacy collection classes such as Vector and Properties.

Legacy classes are those classes that are coming from the first version of Java. Early versions of Java do not include "collections framework". Instead, it defined several classes and one interface for storing objects.
When collections came in the Java 1.2 version, several of the original classes were re-engineered to support the collection interfaces.

Thus, they are fully compatible with the framework. These old classes are known as legacy classes. The legacy classes defined by java.util are Vector, Hashtable, Properties, Stack, and Dictionary. There is one legacy interface called Enumeration.

Enumeration is read-only. You can just read data from the vector. You cannot remove it from the vector using Enumeration.

Since enumeration is an interface so we cannot create an object of enumeration directly. We can create an object of enumeration by calling elements() method of the Vector class.

Syntax

public Enumeration elements() // Return type is Enumeration.

For example:

```
Enumeration e = v.elements(); // Here, v is a vector class object.
```

Methods of Enumeration

The Enumeration interface defines the following two methods. They are as follows:

1. public boolean hasMoreElements(): When this method is implemented, hasMoreElements() will return true If there are still more elements to extract and false if all the elements have been enumerated.
2. public Object nextElement(): The nextElement() method returns next element in the enumeration. It will throw NoSuchElementException when the enumeration is complete.

```
import java.util.Enumeration;
import java.util.Vector;
public class EnumerationTest
{
public static void main(String[] args)
{
        // Create object of vector class without using generic.
        Vector v = new Vector();

        // Add ten elements of integer type using addElement() method.
         for(int i = 0; i < = 5; i++) {
            v.addElement(i);
           }
           System.out.println(v);//print all elements at a time[0, 1, 2, 3, 4, 5]
        // Get elements one by one. So, will require Enumeration concept.
        // Create object of Enumeration by calling elements() method of vector class using object reference variable v.
```

```
        // At the beginning, e (cursor) will point to index just before the first element in v.
        Enumeration e = v.elements();

        // Checking the next element availability using reference variable e and while loop.
        while(e.hasMoreElements())
        {
                // Moving cursor to next element.
                Object o = e.nextElement();
                Integer i = (Integer)o; // Here, Type casting is required because the return type of nextElement() method is an object. Therefore, it's compulsory to require type casting.
                System.out.println(i);
        }
    }
}
```

Limitation of Enumeration

There are many limitations of using enumeration interface in java. They are as follows:

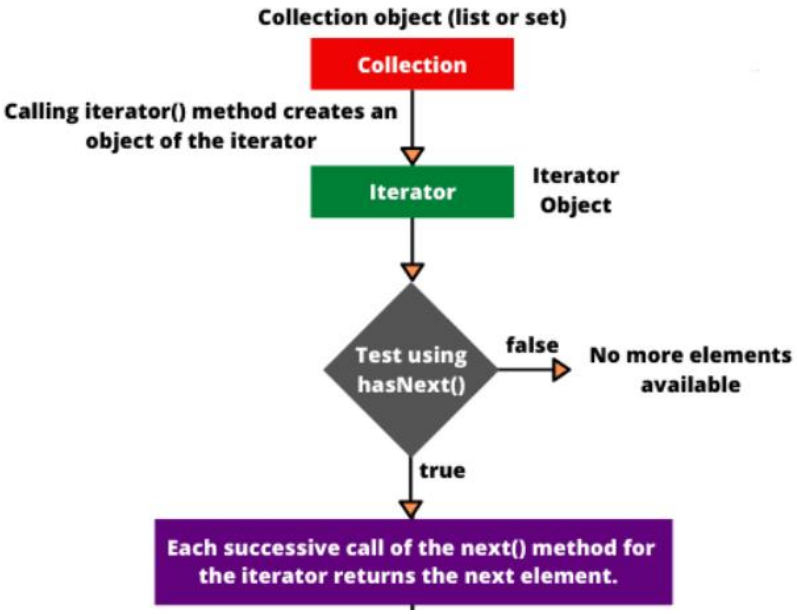
1. Enumeration concept is applicable for only legacy class. Hence, it is not a universal cursor.
2. We can get only read operation by using the enumeration. We cannot perform the remove operation.
3. We can iterate using enumeration only in the forward direction.
4. Java is not recommended to use enumeration in new projects.

To overcome these limitations, We should go for the next level Iterator concept in Java.

Iterator

Iterator in Java is used in the Collections Framework to retrieve elements sequentially (one by one). It is called **universal Iterator** or cursors.
It can be applied to any collection object. By using Iterator, we can perform both read and remove operations.

How Java Iterator works internally?



Difference between Enumeration and Iterator

Both are useful to retrieve elements from a collection. But the main difference between Enumeration and iterator is with respect to functionality.

By using an enumeration, we can perform only read access but using an iterator, we can perform both read and remove operation.

Advantage of Iterator in Java

- An iterator can be used with any collection classes.
- We can perform both read and remove operations.
- It acts as a universal cursor for collection API.

Limitation of Iterator in Java

- By using Enumeration and Iterator, we can move only towards forwarding direction. We cannot move in the backward direction. Hence, these are called single-direction cursors.
- We can perform either read operation or remove operation.
- We cannot perform the replacement of new objects.
- For example, suppose there are five Apple in a box. Out of five, two Apple are not good but we cannot replace those damaged one with new Apple.

Solution is to use ListIterator

ListIterator in Java is the most powerful iterator or cursor that was introduced in Java 1.2 version. It is a bi-directional cursor.
Java ListIterator is an interface (an extension of Iterator interface) that is used to retrieve the elements from a collection object in both forward and reverse directions.

Java ListIterator can be used for all List implemented classes such as ArrayList, CopyOnWriteArrayList, LinkedList, Stack, Vector, etc.

Methods of ListIterator in Java

Forward direction:

1. public boolean hasNext(): This method returns true if the ListIterator has more elements when iterating the list in the forward direction.
2. public Object next(): This method returns the next element in the list. The return type of next() method is Object.
3. public int nextIndex(): This method returns the index of the next element in the list. The return type of this method is an integer.

Backward direction:

4. public boolean hasPrevious(): It checks that list has more elements in the backward direction. If the list has more elements, it will return true. The return type is boolean.
5. public Object previous(): It returns the previous element in the list and moves the cursor position backward direction. The return type is Object.
6. public int previousIndex(): It returns the index of the previous element in the list. The return type is an Integer.

Other capability methods:

7. public void remove(): This method removes the last element returned by next() or previous() from the list. The return type is 'nothing'.
8. public void set(Object o): This method replaces the last element returned by next() or previous() with the new element.
9. public void add(Object o): This method is used to insert a new element in the list.

Example:

```
import java.util.LinkedList;
```

```
import java.util.List;
import java.util.ListIterator;
public class ListIteratorTest {
public static void main(String[] args)
{
List<String> list = new LinkedList<>();
   list.add("A");
   list.add("B");
   list.add("C");
// Creating ListIterator object.
   ListIterator<String> listIterator = list.listIterator();

// Traversing elements in forwarding direction.
  System.out.println("Forward Direction Iteration:");
while(listIterator.hasNext()) {
  System.out.println(listIterator.next());
}
// Traversing elements in the backward direction. The ListIterator cursor is at just after the last element.
   System.out.println("Backward Direction Iteration:");
while(listIterator.hasPrevious()) {
  System.out.println(listIterator.previous());
}
}
}
```

Output:

Forward Direction Iteration:

A

B

C

Backward Direction Iteration:

C

B

A

ListIterator is the most powerful cursor but it still has some limitations. They are as follows:

1. Java List Iterator is applicable only for list implemented class objects. Therefore, it is not a universal Java cursor.
2. It is not applicable to whole collection API.

Iterator	ListIterator
1. Java Iterator is applicable to the whole Collection API.	1. Java ListIterator is only applicable for List implemented classes such as ArrayList, CopyOnWriteArrayList, LinkedList, Stack, Vector, etc.
2. It is a Universal Iterator.	2. It is not a Universal Iterator in Java.
3. Iterator supports only forward direction Iteration.	3. ListIterator supports both forward and backward direction iterations.
4. It is known as a uni-directional iterator.	4. It is also known as bi-directional iterator.
5. Iterator supports only read and delete operations.	5. ListIterator supports all the operations such as read, remove, replacement, and the addition of the new elements.
6. We can get the Iterator object by calling iterator() method.	6. We can create ListIterator object by calling listIterator() method.

Differences Between Enumeration and Iterator In Java

1) Introduction

Iterator interface is introduced from JDK 1.2 where as *Enumeration* interface is there from JDK 1.0.

2) remove() method

Enumeration only traverses the *Collection* object. You can't do any modifications to *Collection* while traversing the *Collection* using *Enumeration*. Where as *Iterator* interface allows us to remove an element while traversing the *Collection* object.

Iterator has *remove()* method which is not there in the *Enumeration* interface.

3) Fail-Fast Vs Fail-Safe

Iterator is a fail-fast in nature. i.e it throws *ConcurrentModificationException* if a collection is modified while iterating other than it's own *remove()* method. Where as *Enumeration* is fail-safe in nature. It doesn't throw any exceptions if a collection is modified while iterating.

Set

Java Map Interface