6.2 Interfaces- List, Set

1) List Interface:

A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for the following:

Positional access — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.

Search — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.

Range-view — The sublist method performs arbitrary range operations on the list.

The Java platform contains two general-purpose List implementations. ArrayList, which is usually the better-performing implementation, and LinkedList which offers better performance under certain circumstances.

Example of List:

```
import java.util.List;
import java.util.ArrayList;
import java.util.LinkedList;

public class ListExample
{
	public static void main(String[] args)
	{
		List<String> al = new ArrayList<String>();
		al.add("BMW");
		al.add("Audi");
		al.add("BMW");

		System.out.println("List Elements: ");
		System.out.print(al);
	}
}
```

Output:

```
List Elements:
[BMW, Audi, BMW]
```

2) Set Interface:

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.

The Java platform contains three general - purpose Set implementations: HashSet, TreeSet, and LinkedHashSet.

HashSet, which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration.

TreeSet, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet.

LinkedHashSet, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).

Example of Set:

```
import java.util.Set;
import java.util.HashSet;
import java.util.TreeSet;
public class SetExample
{
	public static void main(String args[])
	{
		int count[] = {2, 4, 3, 5};
		Set<Integer> hset = new HashSet<Integer>();
		try
		{
			for(int i = 0; i<4; i++)
			{
				hset.add(count[i]);
			}
			System.out.println(hset);
		}
		catch(Exception e)
		{
			e.printStackTrace();
		}
	}
}
```

Output:

```
[2, 4, 3, 5]
```

Difference between List and Set in Java

Sr. No.	Key	List	Set
1	Positional access	The list provides positional access of the elements in the collection.	Set doesn't provide positional access to the elements in the collection
2	Implementation	Implementation of List are ArrayList, LinkedList, Vector ,Stack	Implementation of a set interface is HashSet and LinkedHashSet
3	Duplicate	We can store the duplicate elements in the list.	We can't store duplicate elements in Set
4	Ordering	List maintains insertion order of elements in the collection	Set doesn't maintain any order
5	Null Element	The list can store multiple null elements	Set can store only one null element

6.3 Classes- ArrayList, Vector

1) ArrayList Class:

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. The syntax is also slightly different:

e.g:

Create an ArrayList object called **cars** that will store strings:

```
import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Add Items

The ArrayList class has many useful methods. For example, to add elements to the ArrayList, use the add() method:

e.g:

```
import java.util.ArrayList;

public class arraylst
{
	public static void main(String[] args)
	{
		ArrayList<String> cars = new ArrayList<String>();
		cars.add("Volvo");
		cars.add("BMW");
		cars.add("Ford");
		cars.add("Mazda");
		System.out.println(cars);
	}
}
```

Access an Item

To access an element in the ArrayList, use the get() method and refer to the index number:

e.g:

```
cars.get(0);
```

Change an Item

To modify an element, use the set() method and refer to the index number:

e.g:

```
cars.set(0, "Opel");
```

Remove an Item

To remove an element, use the remove() method and refer to the index number:

e.g:

```
cars.remove(0);
```

To remove all the elements in the ArrayList, use the clear() method:

e.g:

```
cars.clear();
```

ArrayList Size

To find out how many elements an ArrayList have, use the size method:

e.g:

```
cars.size();
```

Loop Through an ArrayList

Loop through the elements of an ArrayList with a for loop, and use the size() method to specify how many times the loop should run:

e.g:

```
public class arraylst
```

```
{
	public static void main(String[] args)
	{
		ArrayList<String> cars = new ArrayList<String>();
		cars.add("Volvo");
		cars.add("BMW");
		cars.add("Ford");
		cars.add("Mazda");

		for (int i = 0; i < cars.size(); i++)
		{
			System.out.println(cars.get(i));
		}
	}
}
```

2) Vector Class:

Vector implements List Interface. Like ArrayList it also maintains insertion order but it is rarely used in non-thread environment as it is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.

Three ways to create vector class object:

Method 1:

```
Vector vec = new Vector();
```

It creates an empty Vector with the default initial capacity of 10. It means the Vector will be re-sized when the 11th elements needs to be inserted into the Vector. Note: By default vector doubles its size. i.e. In this case the Vector size would remain 10 till 10 insertions and once we try to insert the 11th element It would become 20 (double of default capacity 10).

Method 2:

Syntax: Vector object= new Vector(int initialCapacity)

e.g: Vector vec = new Vector(3);

It will create a Vector of initial capacity of 3.

Method 3:

Syntax:

Vector object= new vector(int initialcapacity, capacityIncrement)

Example: Vector vec= new Vector(4, 6)

Here we have provided two arguments. The initial capacity is 4 and capacity Increment is 6. It means upon insertion of 5th element the size would be 10 (4+6) and on 11th insertion it would be 16(10+6).

Complete Example of Vector in Java:

```
import java.util.*;

public class VectorExample
{
	public static void main(String args[])
	{
		/* Vector of initial capacity(size) of 2 */
		Vector<String> vec = new Vector<String>(2);

		/* Adding elements to a vector*/
		vec.addElement("Apple");
		vec.addElement("Orange");
		vec.addElement("Mango");
		vec.addElement("Fig");

		/* check size and capacityIncrement*/
		System.out.println("Size is: "+vec.size());
		System.out.println("Default capacity increment is: "+vec.capacity());

		vec.addElement("fruit1");
		vec.addElement("fruit2");
		vec.addElement("fruit3");

		/*size and capacity Increment after two insertions*/
		System.out.println("Size after addition: "+vec.size());
		System.out.println("Capacity after increment is: "+vec.capacity());

		/*Display Vector elements*/
		Enumeration en = vec.elements();
		System.out.println("\nElements are:");
		while(en.hasMoreElements())
		{
			System.out.print(en.nextElement() + " ");
		}
  }
}
```

Output:

```
Size is: 4
Default capacity increment is: 4
Size after addition: 7
Capacity after increment is: 8

Elements are:
Apple Orange Mango Fig fruit1 fruit2 fruit3
```