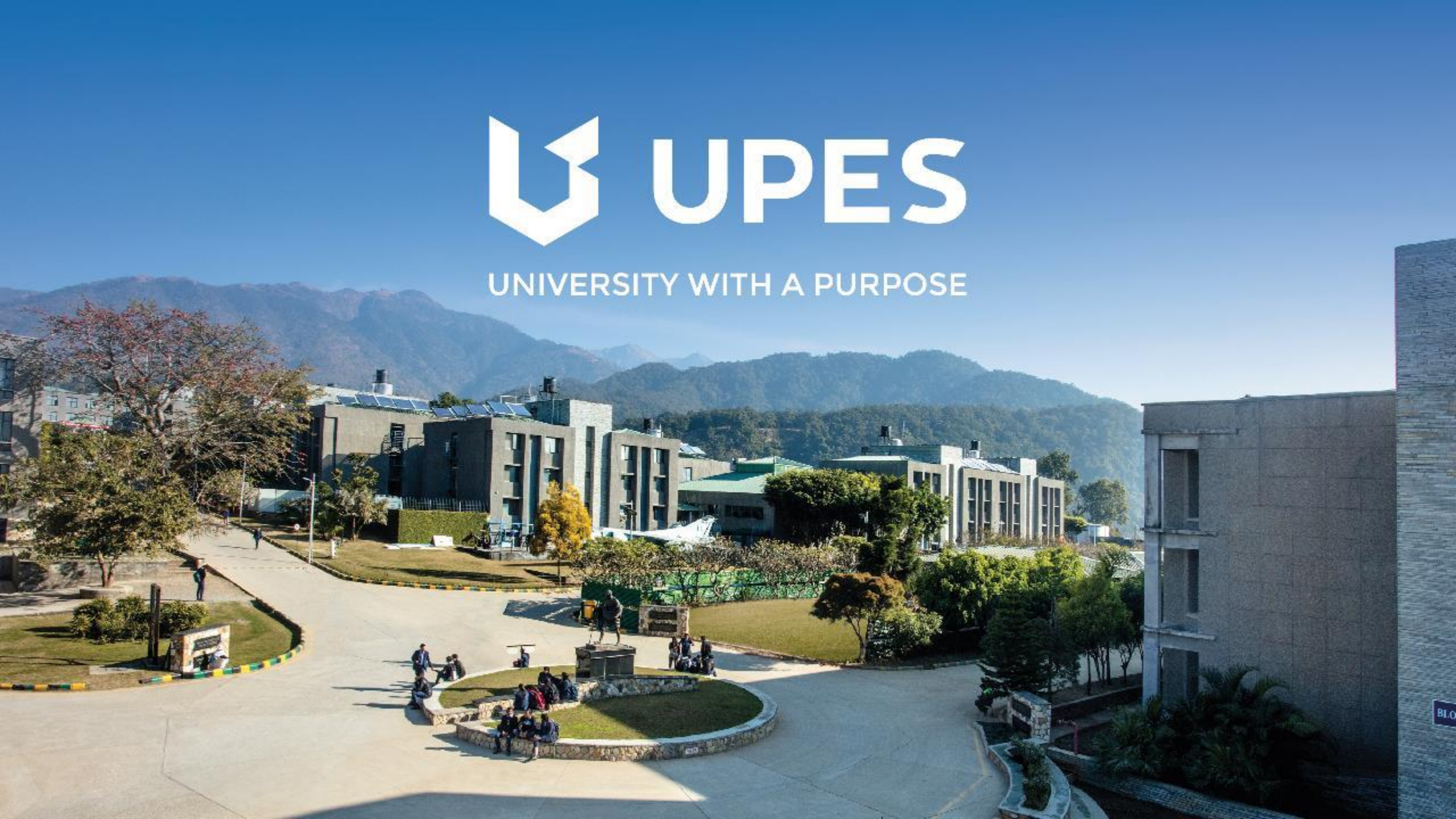
UPES
UNIVERSITY WITH A PURPOSE

# LinkedList, Vector, and Stack

# Collection Framework

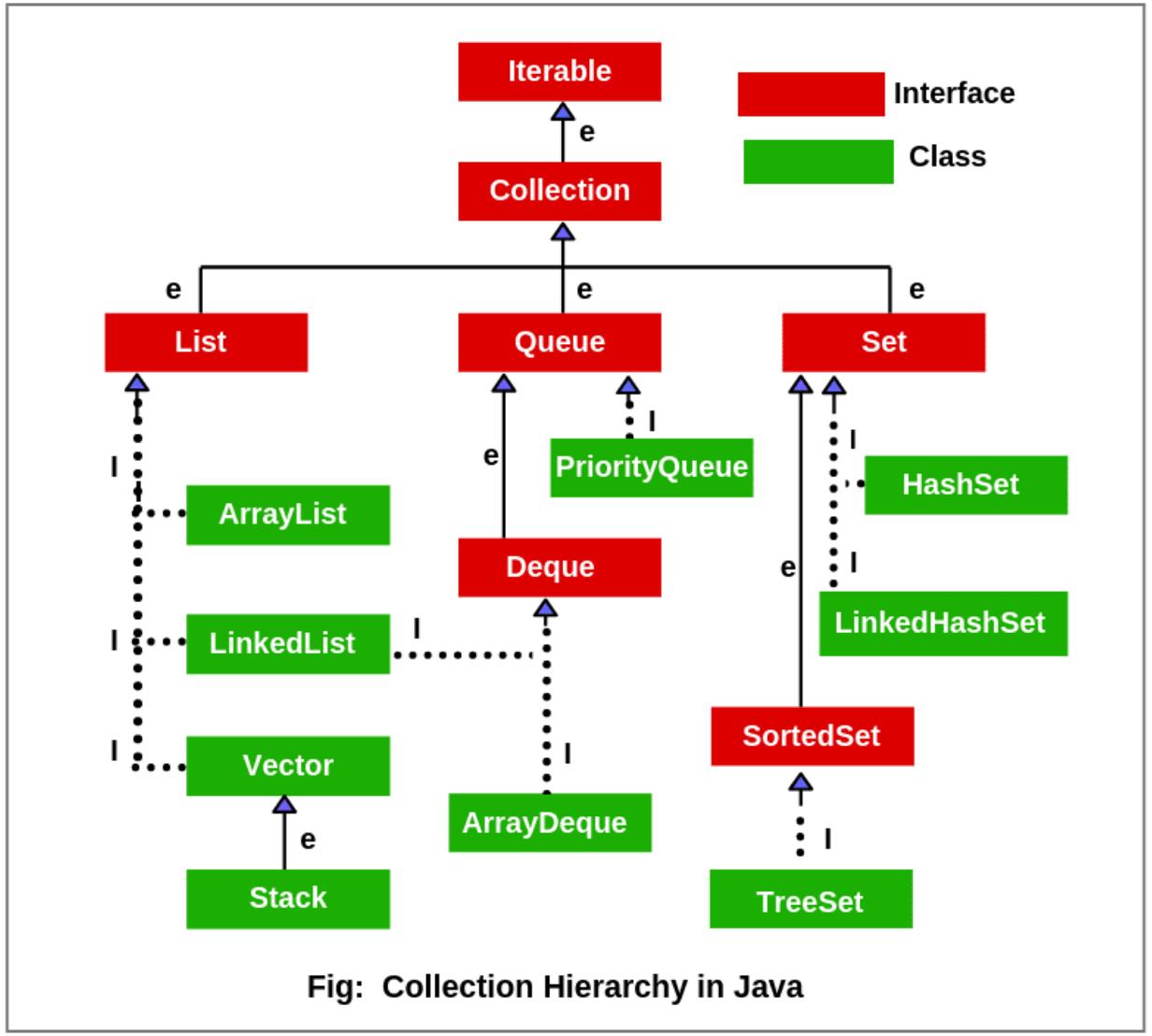


Fig: Collection Hierarchy in Java

# ArrayList and LinkedList

- In ArrayList, elements are stored in consecutive memory locations.
- Insertion in the middle requires massive shifting operations, which may affect the efficiency.
- In linkedList, elements are not necessary to be stored in contiguous memory locations. Cost of inserting an element in the middle is lesser as no shift operations are needed.
- Accessing nth element in the LinkedList is costly.
- Accessing elements in ArrayList takes constant time.

# Methods of LinkedList

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

1) **boolean add(Object item)**: It adds the item at the end of the list.

```
llistobj.add("Hello");          //It would add the string “Hello” at the end of the linked list.
```

**2) void add(int index, Object item):** It adds an item at the given index of the the list.

```
llistobj.add(2, "bye");         //This will add the string “bye” at the 3rd position
```

3) **boolean addAll(Collection c)**: It adds all the elements of the specified collection c to the list. It throws NullPointerException if the specified collection is null.

```
LinkedList<String> llistobj = new LinkedList<String>();
ArrayList<String> arraylist= new ArrayList<String>();
arraylist.add("String1");
arraylist.add("String2");
llistobj.addAll(arraylist);
//This piece of code would add all the elements of ArrayList to the LinkedList.
```

# Methods of LinkedList (cont..)

4) **boolean addAll(int index, Collection c):** It adds all the elements of collection c to the list starting from a give index in the list. It throws NullPointerException if the collection c is null and IndexOutOfBoundsException when the specified index is out of the range.
llistobj.add(5, arraylist); //It would add all the elements of the ArrayList to the LinkedList starting from position 6 (index 5).

**5) void addFirst(Object item):** It adds the item (or element) at the first position in the list.
llistobj.addFirst("text"); //It would add the string “text” at the beginning of the list.

**6) void addLast(Object item):** It inserts the specified item at the end of the list.
llistobj.addLast(“Text1"); //This statement will add a string “Text1” at the end position of the linked list.

# Methods of LinkedList (cont..)

**7) void clear():** It removes all the elements of a list.
llistobj.clear();

**8) boolean contains(Object item):** It checks whether the given item is present in the list or not. If the item is present then it returns true else false.
boolean var = llistobj.contains("TestString"); //It will check whether the string “TestString” exist in the list or not.

**9) Object get(int index):** It returns the item of the specified index from the list.
Object var = llistobj.get(2); //It will fetch the 3rd item from the list.

**10) Object getFirst():** It fetches the first item from the list.
Object var = llistobj.getFirst();

**11) Object getLast():** It fetches the last item from the list.
Object var= llistobj.getLast();

# Methods of LinkedList (cont..)

**12) int indexOf(Object item):** It returns the index of the specified item.
llistobj.indexOf("bye");

**13) int lastIndexOf(Object item):** It returns the index of last occurrence of the specified element.
int pos = llistobj.lastIndexOf(“hello”);  //integer variable pos will be having the index of last occurrence of string “hello”.

**14) Object remove():** It removes the first element of the list.
llistobj.remove();

**15) Object remove(int index):** It removes the item from the list which is present at the specified index.
llistobj.remove(4);  //It will remove the 5th element from the list.

**16) Object remove(Object obj):** It removes the specified object from the list.
llistobj.remove("Test Item");

# Methods of LinkedList (cont..)

**17) Object removeFirst():** It removes the first item from the list.
llistobj.removeFirst();

**18) Object removeLast():** It removes the last item of the list.
llistobj.removeLast();

**19) Object removeFirstOccurrence(Object item):** It removes the first occurrence of the specified item.
llistobj.removeFirstOccurrence("text");

**20) Object removeLastOccurrence(Object item):** It removes the last occurrence of the given element.
llistobj.removeLastOccurrence(“String1”);

**21) Object set(int index, Object item):** It updates the item of specified index with the give value.
llistobj.set(2, "Test");

**22) int size():** It returns the number of elements of the list.
llistobj.size();

# Examples

```
import java.util.*;
public class LinkedList1
{
    public static void main(String[] args)
    {
        LinkedList list = new LinkedList();

        list.addFirst("Lockdown");

        list.addLast("In India");

         System.out.println(list.getFirst());

       System.out.println(list.getLast());

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

       System.out.println(list.get(1));
    }
}
```

# Vector

**All Implemented Interfaces:**
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

**Direct Known Subclasses:**
Stack

- The Vector class implements a growable array of objects.
- Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.
- Each vector tries to optimize storage management by maintaining a capacity and a capacityIncrement.
- The capacity is always at least as large as the vector size; it is usually larger because as components are added to the vector, the vector's storage increases in chunks the size of capacityIncrement.

# Vector Constructors

| Constructor | Description |
| --- | --- |
| **Vector**() | Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero. |
| **Vector**(int initialCapacity) | Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero. |
| **Vector**(int initialCapacity, int capacityIncrement) | Constructs an empty vector with the specified initial capacity and capacity increment. |
| **Vector**(**Collection** c) | Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator. |

# Vector and ArrayList

- ArrayList is not threadsafe as its methods are non-synchronized. High performance. ArrayList was first introduced in version 1.2

- Vector is threadsafe. Vector methods are synchronized. Low performance. First introduced in version 1.0 and then reengineered in version 1.2

- Unlike the new collection implementations, Vector is synchronized. If a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector.

# References

http://java.sun.com/docs/books/tutorial/collections/interfaces/collection.html.

# THANK YOU