



Chapter 1

Collections and Generics

Computer Science and Engineering Department@SoEEC
Advanced Programming(CSE 3312)

Objectives:

At the end of this lecture students will be able to :

- ▮ Describe Java Collection Framework
- ▮ Identify Hierarchy of Collection Framework
- ▮ Identify Collection interfaces and Classes
- ▮ Use Iterator Interface and Generic Classes
- ▮ Use commonly used collection algorithms

Introduction

- In order to handle group of objects we can use array of objects. If we have a class called Employ with members name and id, if we want to store details of 10 Employees, create an array of object to hold 10 Employ details.

Employ ob [] = new Employ [10];

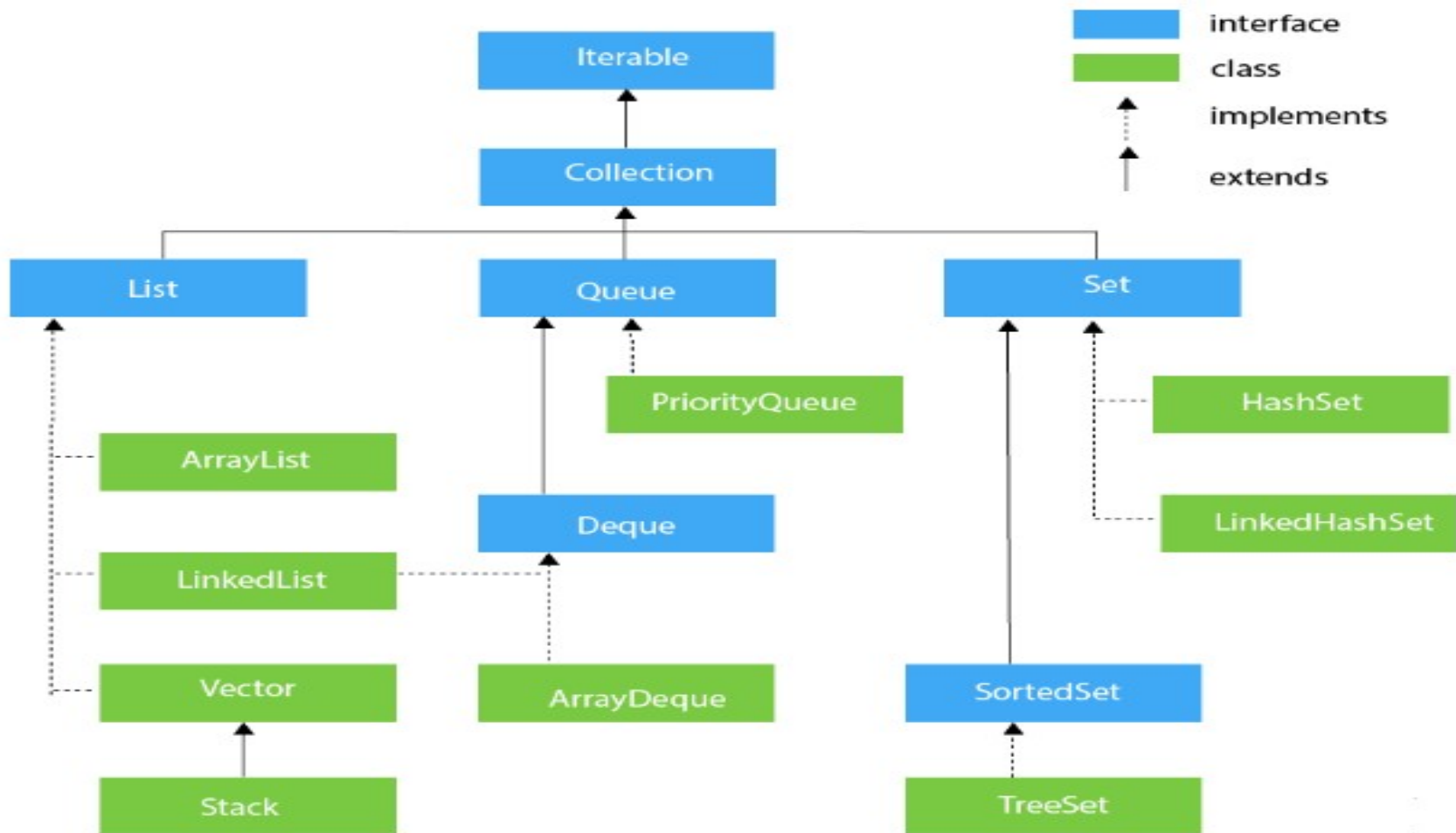
- We cannot store different class objects into same array.
- Inserting element at the end of array is easy but at the middle is difficult.
- After retrieving the elements from the array, in order to process the elements we don't have any methods

Java Collections Framework

- It is a Pre packaged Implementation
- A Java collection framework provides an architecture to store and manipulate a group of objects.
- The JCF provides developers to access pre-packaged data structures as well as algorithms to manipulate data
- Java collections framework includes the following:
 - **Interfaces:**
 - are abstract data types that represent collections.
 - allow collections to be manipulated independently of the details of their representation.
 - **Implementations, i.e., Classes:**
 - are the implementation of the collection interfaces.
 - are reusable data structures.
 - **Algorithm:**
 - the methods which are used to perform operations such as searching and sorting, on objects that implement collection interfaces
 - are said to be **polymorphic**: that is, the same method can be used on many different implementations of the appropriate collection interface.

Hierarchy of Collection Framework

- Collections are used to store, retrieve, manipulate, and communicate aggregate data.



.....Contd

- **Iterable:** The root interface of the collection hierarchy. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
- **Collection:** 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. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.
- **List:** 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. List interface is implemented by the classes [ArrayList](#), [LinkedList](#), [Vector](#), and [Stack](#).
- **Set:** Set interface 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](#).
- **Queue:** 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.

....Contd

- **SortedSet:** 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.
- **Map:** It represents key-value pairs. The keys will be unique and each key can map to at most one value. Although, it does not ensure element ordering, some implementations guarantee it. To interoperate with other collection classes/interfaces, it provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.

Collection

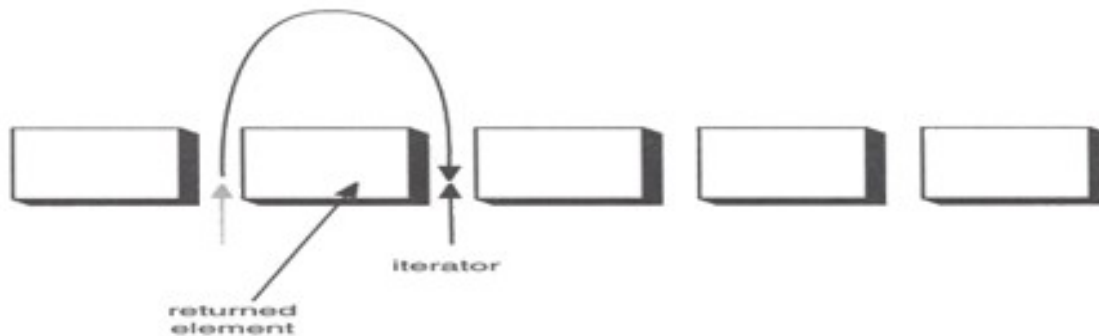
- A collection is a container that encapsulates multiple objects into a single unit, such as a list of employees, a set of numbers, a set of, processes, a queue of requests etc..
 - All the collection classes are available in the package called 'java.util'
 - Group of collection classes is called a Collection Framework.
- Some examples of collections are
 - the cards you hold in a card game,
 - your favorite songs stored in your computer,
 - the members of a sports team and
 - the real-estate records in your local registry of deeds (which map book numbers and page

Collection Interface

- Defines fundamental methods
 - **int size();**
 - **boolean isEmpty();**
 - **boolean contains(Object element);**
 - **boolean add(Object element);**
 - **boolean remove(Object element);**
 - **Iterator iterator();**
- These methods are enough to define the basic behavior of a collection
- Provides an Iterator to step through the elements in the Collection

Iterator Interface

- Iterator is an interface that iterates the elements. It is used to traverse the list and modify the elements. Traverse forward direction only.
- Defines three fundamental methods
 - **public boolean hasNext()** - This method returns true if the iterator has more elements.
 - **public Object next()** - It returns the element and moves the cursor pointer to the next element.
 - **public void remove()** - This method removes the last elements returned by the iterator.
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection



the collection

Example

```
public class SimpleCollection {  
    public static void main(String[] args) {  
        Collection c;  
        c = new ArrayList();
```

```
        System.out.println(c.getClass().getName());
```

```
        for (int i=1; i <= 10; i++) {
```

```
            c.add(i + " * " + i + " = " + i*i);
```

```
        }
```

```
        Iterator iter = c.iterator();
```

```
        while (iter.hasNext())
```

```
            System.out.println(iter.next());
```

```
    }
```

```
}
```

Output:

java.util.ArrayList

1 * 1 = 1

2 * 2 = 4

3 * 3 = 9

4 * 4 = 16

5 * 5 = 25

6 * 6 = 36

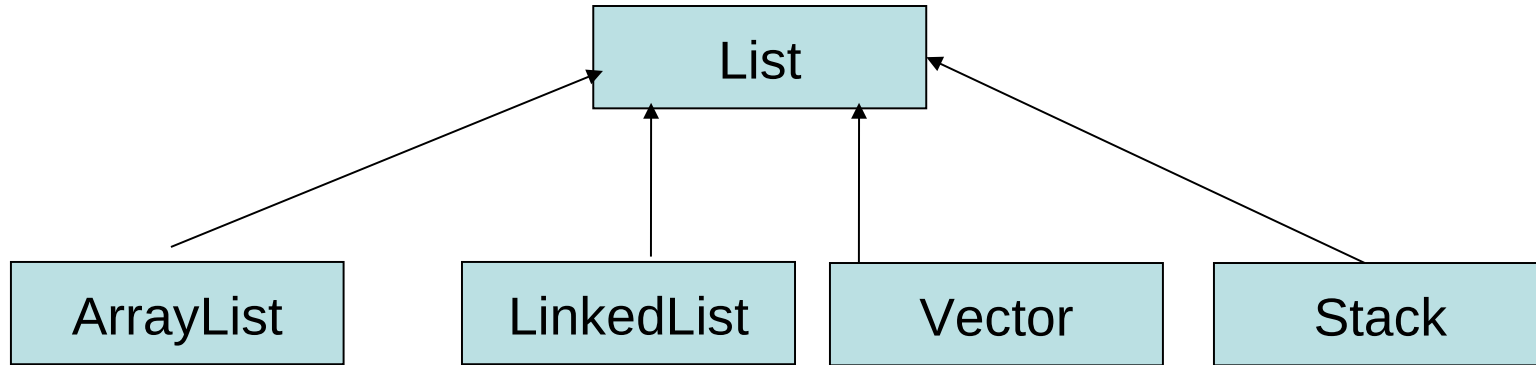
7 * 7 = 49

8 * 8 = 64

9 * 9 = 81

10 * 10 = 100

List Interface



- A List is an ordered Collection of elements which may contain duplicates. It is an interface that extends the Collection interface.
- The List interface adds the notion of *order* to a collection.
- The user of a list has control over where an element is added in the collection.

The Common Methods in the List Interface

«interface»
java.util.Collection<E>



«interface»
java.util.List<E>

```
+add(index: int, element: Object): boolean  
+addAll(index: int, c: Collection<? extends E>  
: boolean  
+get(index: int): E  
+indexOf(element: Object): int  
+lastIndexOf(element: Object): int  
+listIterator(): ListIterator<E>  
+listIterator(startIndex: int): ListIterator<E>  
+remove(index: int): E  
+set(index: int, element: Object): Object  
+subList(fromIndex: int, toIndex: int): List<E>
```

Adds a new element at the specified index.

Adds all the elements in `c` to this list at the specified index.

Returns the element in this list at the specified index.

Returns the index of the first matching element.

Returns the index of the last matching element.

Returns the list iterator for the elements in this list.

Returns the iterator for the elements from `startIndex`.

Removes the element at the specified index.

Sets the element at the specified index.

Returns a sublist from `fromIndex` to `toIndex-1`.

ListIterator Interface

- Extends the Iterator interface
- Defines three fundamental methods
 - **void add(Object o)**
 - **boolean hasPrevious()**
 - **Object previous()**
- The addition of these three methods defines the basic behavior of an ordered list
- A ListIterator knows position within list.

ArrayList

- ArrayList is the implementation of List Interface where the elements can be dynamically added or removed from the list. Also, the size of the list is increased dynamically if the elements are added more than the initial size.
- Important Points:
 - Java ArrayList class can contain duplicate elements.
 - Java ArrayList class maintains insertion order.
 - Java ArrayList class is non synchronized.
 - Java ArrayList allows random access because array works at the index basis.
 - In Java ArrayList class, manipulation is slow because a lot of shifting needs to occur if any element is removed from the array list.

ArrayList Methods

- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
 - **Object get(int index)**
 - **Object set(int index, Object element)**
- Indexed add and remove are provided, but can be costly if used frequently
 - **void add(int index, Object element)**
 - **Object remove(int index)**
- May want to resize in one shot if adding many elements
 - **void ensureCapacity(int minCapacity)**

Example: ArrayList

```
import java.util.*;
```

```
class ArrayListDemo
```

```
{ public static void main(String args[])
```

```
    { ArrayList <String> al = new ArrayList<String>();
```

```
      al.add ("Sheger"); al.add ("Adama");
```

```
      al.add ("Hawassa"); al.add ("BahirDar");
```

```
      al.add ("Mekele");
```

```
      al.add (1,"DireDawa");
```

```
      al.add (2,"Asosa");
```

```
      System.out.print("Size of the Array List is: " + al.size ());
```

```
      System.out.print("\nRetrieving elements in ArrayList using Iter
```

```
      Iterator it = al.iterator ();
```

```
      while (it.hasNext () )
```

```
          System.out.print (it.next () + "\t");
```

```
    } }
```

LinkedList

- This class implements two collection interfaces, List and Deque and provides all operations of these two interfaces. It inherits the AbstractList class and implements List and Deque interfaces.
- It uses a doubly linked list internally to store the elements.
- It provides a linked-list data structure.
- Important points:
 - LinkedList class can contain duplicate elements.
 - LinkedList class maintains insertion order.
 - It is non synchronized.
 - Its manipulation is fast because no shifting needs to occur.

LinkedList Methods

`java.util.AbstractSequentialList<E>`



`java.util.LinkedList<E>`

```
+LinkedList()  
+LinkedList(c: Collection<? extends E>)  
+addFirst(element: E): void  
+addLast(element: E): void  
+getFirst(): E  
+getLast(): E  
+removeFirst(): E  
+removeLast(): E
```

Creates a default empty linked list.

Creates a linked list from an existing collection.

Adds the element to the head of this list.

Adds the element to the tail of this list.

Returns the first element from this list.

Returns the last element from this list.

Returns and removes the first element from this list.

Returns and removes the last element from this list.

Example: LinkedList

```
import java.util.*;
public class LinkedList1 {
    public static void main(String args[]){
        LinkedList<String> pl=new LinkedList<String>();
        pl.add("Java");
        pl.add("Python");
        pl.add("C++");
        pl.add("C#");
        Iterator<String> itr=pl.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        } } }
```

Vector

- Vectors are similar to arrays, where the elements of the vector object can be accessed via an index into the vector. Vector implements a dynamic array. Also, the vector is not limited to a specific size, it can shrink or grow automatically whenever required. It is similar to ArrayList, but with two differences :
 - Vector is synchronized.
 - Vector contains many legacy methods that are not part of the

• So

Methods	Description
int size()	Returns the total number of elements in Vector.
boolean isEmpty()	Returns true if Vector is empty.
E firstElement()	Returns the first element in Vector.
E lastElement()	Returns the last element in Vector.
E elementAt(int index)	Returns the element present in Vector at a specified index.
void addElement(E element)	Inserts an element in the Vector.
boolean contains(Object object)	Returns true if the specified object is present in Vector.
boolean removeElementAt(int index)	Returns true if element present in Vector is removed.

```

import java.util.*;
public class Vector1{
public static void main(String args[]){
Vector<String> v=new Vector<String>(
);

```

```

v.add("SoEEC");
v.add("SoMMCE");
v.add("SoCEA");
v.add("SoANC");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}

```

```

import java.util.*;
public class LegVector1 { public static
void main(String ar[])
{ Vector<String> vec= new
Vector<String>(); vec.addElement("B");
vec.addElement("A"); vec.addElement("T");
vec.addElement("M"); vec.addElement("A");
vec.addElement("N");
System.out.println("Vector = "+vec);
System.out.println("Size of Vector = "+
vec.size()); System.out.println("First element
in Vector = "+vec.firstElement());
System.out.println("Last element in Vector
= "+vec.lastElement());
System.out.println("Element at 2nd index =
"+ vec.elementAt(2));
vec.removeElementAt(2);
System.out.println("After removing element

```

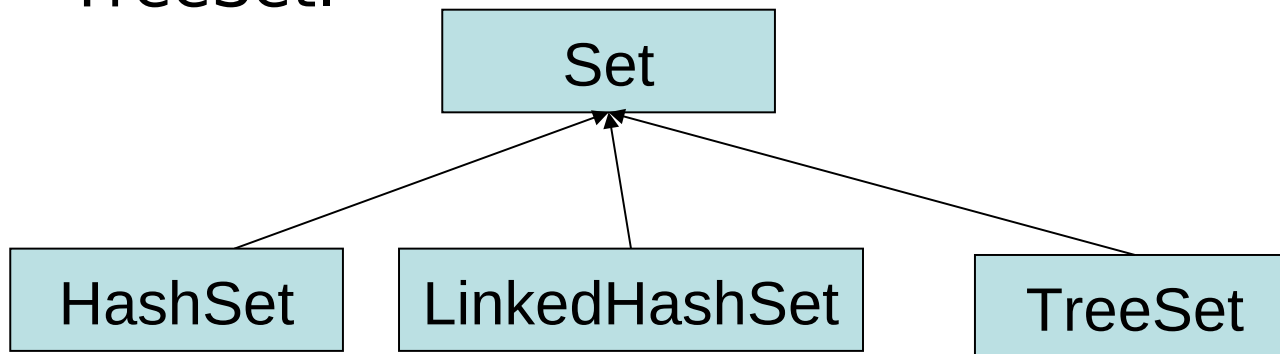
Stack

- The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean pop(), boolean peek() which defines its properties.

```
import java.util.*;  
public class TestJavaCollection4{  
  public static void main(String args[]){  
    Stack<String> stack = new Stack<String>();  
    stack.push("SoEEC");  
    stack.push("SoMMCE");  
    stack.push("SoCEA");  
    stack.push("SoANC");  
    stack.push("SoHSS");  
    stack.pop();  
    Iterator<String> itr=stack.iterator();  
    while(itr.hasNext()){  
      System.out.println(itr.next());  
    }  
  }  
}
```

Set Interface

- Same methods as Collection
 - No duplicate entries
 - Implemented by HashSet, LinkedHashSet, and TreeSet.



HashSet

- HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.
- stores the elements by using a mechanism called **hashing**.
- contains unique elements only.
- non synchronized.
- doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- best approach for search operations.

HashSet Example

```
import java.util.*;
class HashSetDemo{
    public static void main(String args[]){
        //Creating HashSet and adding elements
        HashSet<String> set=new HashSet();
            set.add("One");
            set.add("Two");
            set.add("Three");
            set.add("Four");
            set.add("Five");
            Iterator<String> i=set.iterator();
            while(i.hasNext()) {
                System.out.println(i.next());
            }
    }
```

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.

```
import java.util.*;
class LinkedHashSet1{
    public static void main(String args[]){
        //Creating HashSet and adding elements
        LinkedHashSet<String> set=new Linke
        dHashSet();
        set.add("One");
        set.add("Two");
        set.add("Three");
        set.add("Four");
        set.add("Five");
        Iterator<String> i=set.iterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

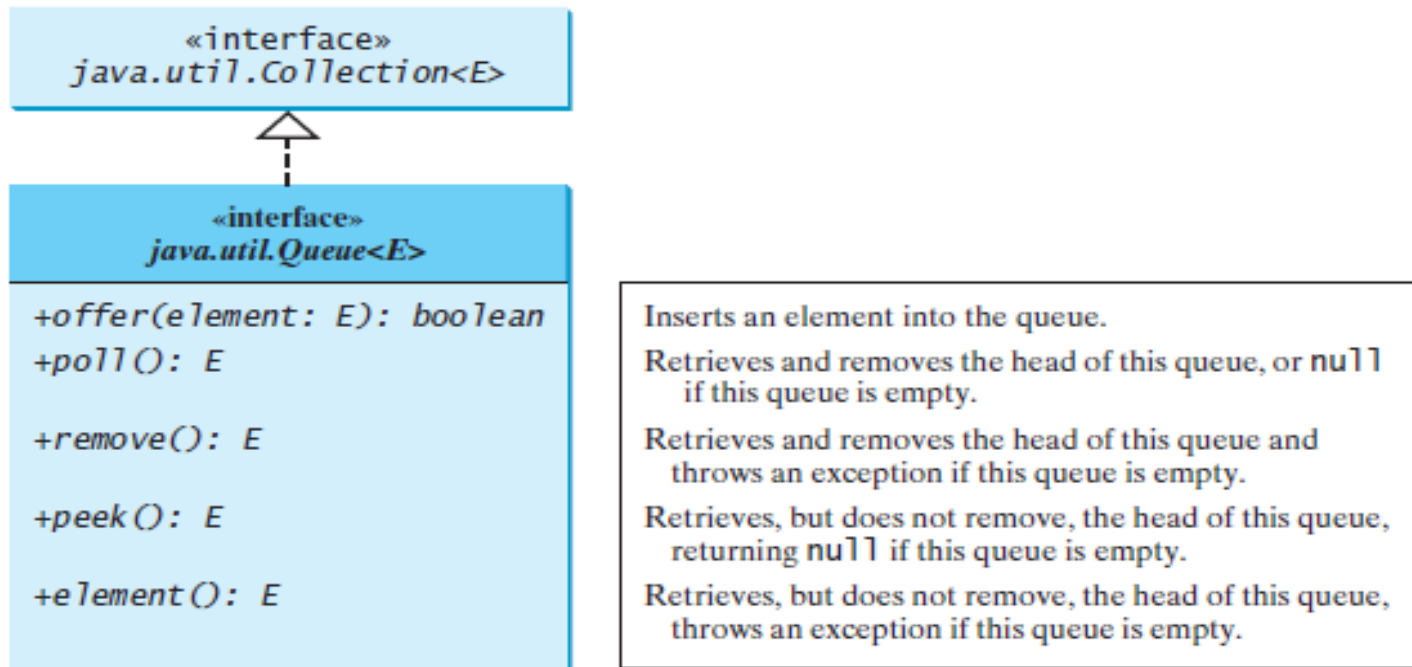
TreeSet

- 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.

```
import java.util.*;  
  
public class TestJavaCollection9{  
  
    public static void main(String args[]) {  
        //Creating and adding elements  
        TreeSet<Integer> set=new TreeSet<>();  
        set.add(5);  
        set.add(2);  
        set.add(1);  
        set.add(3);  
        set.add(4);  
        //traversing elements  
        Iterator<String> itr=set.iterator();
```

Queue Interface

- Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last



PriorityQueue class: The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner.

ArrayDeque class: 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.

Map Interface

- A map contains values on the basis of key, i.e. key and value
- Each key and value pair is known as an entry. A Map contains keys.
- A Map is useful if you have to search, update or delete elements on the basis of a key.
- You can create a map using one of its three concrete classes:

HashMap,

LinkedHashMap, or TreeMap.

Map Methods

«interface»
java.util.Map<K,V>

```
+clear(): void  
+containsKey(key: Object): boolean  
  
+containsValue(value: Object): boolean  
  
+entrySet(): Set<Map.Entry<K,V>>  
+get(key: Object): V  
+isEmpty(): boolean  
+keySet(): Set<K>  
+put(key: K, value: V): V  
+putAll(m: Map<? extends K,? extends V>): void  
+remove(key: Object): V  
+size(): int  
+values(): Collection<V>
```

Removes all entries from this map.

Returns true if this map contains an entry for the specified key.

Returns true if this map maps one or more keys to the specified value.

Returns a set consisting of the entries in this map.

Returns the value for the specified key in this map.

Returns true if this map contains no entries.

Returns a set consisting of the keys in this map.

Puts an entry into this map.

Adds all the entries from *m* to this map.

Removes the entries for the specified key.

Returns the number of entries in this map.

Returns a collection consisting of the values in this map.

Map.Entry interface

- Entry is the subinterface of Map. So, will be accessed it by Map.Entry
- It returns a collection-view of the map, whose elements are of this c
- It provides methods to get key and value.

«interface»
java.util.Map.Entry<K,V>

+getKey(): K
+getValue(): V
+setValue(value: V): void

Returns the key from this entry.
Returns the value from this entry.
Replaces the value in this entry with a new value.

TreeMap Example

```
import java.util.*;
class TreeMap1{
    public static void main(String args[]){
        TreeMap<Integer,String> map=new TreeMap<Integer,String>();
        map.put(1,"CSE");
        map.put(2,"SE");
        map.put(3,"EPCE");
        map.put(4,"ECE");

        for(Map.Entry m:map.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

Generic Class and Methods

- **Generic Class:** A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.
- **Type Parameters:** the commonly type parameters are as follows:
 1. T - Type
 2. E - Element
 3. K - Key
 4. N - Number
 5. V - Value

Creating Generic Class

```
class MyGen<T> {  
    T obj;  
    void add(T obj)  
    {  
        this.obj=obj;  
    }  
    T get()  
    {  
        return obj; }  
}
```

The **T type** indicates that it can refer to any type (like String, Integer, Employee etc.).

The type you specify for the class, will be used to store and retrieve the data.

Using Generic Class

```
public class TestGenerics {  
    public static void main(String args[])  
    {  
        MyGen<Integer> g1=new MyGen<Integer>();  
        g1.add(2);  
        //g1.add("CSE");//Compile time error  
        System.out.println(g1.get());  
    }  
}
```

Generic Method

- Like generic class, we can create generic method that can accept any type of argument. Let's see a simple example of java generic method to print array elements. We are using here **E** to denote the element.

```
public class TestGenericsMethod {  
    public static <E> void printArray(E[] elements){  
        for(E element : elements){  
            System.out.println(element);  
        }  
        System.out.println();  
    }  
    public static void main( String args[] ) {  
        Integer[] intArray = {10,20,30,40,50};  
        Character[] charArray={'C','S','E','-','A','S','T','U'};  
        System.out.println("Printing Integer Array");  
        printArray(intArray);  
        System.out.println("Printing Character Array");  
        printArray(charArray);  
    }  
}
```

Collection Algorithms

- The collections framework defines several algorithms that can be applied to collections and maps.
- These algorithms are defined as static methods within the Collections class.
- The methods defined in collection framework's algorithm are summarized in the following:
 - **static int binarySearch(List list, Object value, Comparator c)**
Searches for value in list ordered according to c. Returns the position of value in list, or -1 if value is not found.
 - **static int binarySearch(List list, Object value)**
Searches for value in list. The list must be sorted. Returns the position of value in list, or -1 if value is not found.
 - **static void copy(List list1, List list2)**
Copies the elements of list2 to list1.
 - **static Enumeration enumeration(Collection c)**
Returns an enumeration over c.
 - **static void fill(List list, Object obj)**
Assigns obj to each element of list.

Contd..

- **static int indexOfSubList(List list, List subList)**
Searches list for the first occurrence of subList. Returns the index of the first match, or .1 if no match is found.
- **static int lastIndexOfSubList(List list, List subList)**
Searches list for the last occurrence of subList. Returns the index of the last match, or .1 if no match is found.
- **static ArrayList list(Enumeration enum)**
Returns an ArrayList that contains the elements of enum.
- **static Object max(Collection c, Comparator comp)**
Returns the maximum element in c as determined by comp.
- **static Object max(Collection c)**
Returns the maximum element in c as determined by natural ordering. The collection need not be sorted.
- **static Object min(Collection c, Comparator comp)**
Returns the minimum element in c as determined by comp. The collection need not be sorted.

Contd..

- **static Object min(Collection c)**

Returns the minimum element in c as determined by natural ordering.

- **static List nCopies(int num, Object obj)**

Returns num copies of obj contained in an immutable list. num must be greater than or equal to zero.

- **static boolean replaceAll(List list, Object old, Object new)**

Replaces all occurrences of old with new in list. Returns true if at least one replacement occurred. Returns false, otherwise.

- **static void reverse(List list)**

Reverses the sequence in list.

- **static Comparator reverseOrder()**

Returns a reverse comparator.

- **static void rotate(List list, int n)**

Rotates list by n places to the right. To rotate left, use a negative value for n.

- **static void shuffle(List list, Random r)**

Shuffles (i.e., randomizes) the elements in list by using r as a

Contd..

- **static void shuffle(List list)**
Shuffles (i.e., randomizes) the elements in list.
- **static Set singleton(Object obj)**
Returns obj as an immutable set. This is an easy way to convert a single object into a set.
- **static List singletonList(Object obj)**
Returns obj as an immutable list. This is an easy way to convert a single object into a list.
- **static Map singletonMap(Object k, Object v)**
Returns the key/value pair k/v as an immutable map. This is an easy way to convert a single key/value pair into a map.
- **static void sort(List list, Comparator comp)**
Sorts the elements of list as determined by comp.
- **static void sort(List list)**
Sorts the elements of list as determined by their natural ordering.
- **static void swap(List list, int idx1, int idx2)**
Exchanges the elements in list at the indices specified by idx1 and idx2.

Contd..

- **static Collection synchronizedCollection(Collection c)**
Returns a thread-safe collection backed by c.
- **static List synchronizedList(List list)**
Returns a thread-safe list backed by list.
- **static Map synchronizedMap(Map m)**
Returns a thread-safe map backed by m.
- **static Set synchronizedSet(Set s)**
Returns a thread-safe set backed by s.
- **static SortedMap synchronizedSortedMap(SortedMap sm)**
Returns a thread-safe sorted set backed by sm.
- **static SortedSet synchronizedSortedSet(SortedSet ss)**
Returns a thread-safe set backed by ss.
- **static Collection unmodifiableCollection(Collection c)**
Returns an unmodifiable collection backed by c.
- **static List unmodifiableList(List list)**
Returns an unmodifiable list backed by list.

Contd..

- **static Map unmodifiableMap(Map m)**
Returns an unmodifiable map backed by m.
- **static Set unmodifiableSet(Set s)**
Returns an unmodifiable set backed by s.
- **static SortedMap
unmodifiableSortedMap(SortedMap sm)**
Returns an unmodifiable sorted map backed by sm.
- **static SortedSet
unmodifiableSortedSet(SortedSet ss)**
Returns an unmodifiable sorted set backed by ss.

Algorithms: Example

```
public class AlgorithmsDemo
```

```
{ public static void main(String args[])
```

```
{ // Create and initialize linked list
```

```
    LinkedList ll = new LinkedList();
```

```
    ll.add(new Integer(-8));
```

```
    ll.add(new Integer(20));
```

```
    ll.add(new Integer(-20));
```

```
    ll.add(new Integer(8));
```

```
    // Create a reverse order comparator
```

```
    Comparator r = Collections.reverseOrder();
```

```
    // Sort list by using the comparator
```

```
    Collections.sort(ll, r);
```

```
    // Get iterator
```

```
    Iterator li = ll.iterator();
```

```
    System.out.print("List sorted in reverse: ");
```

```
    while(li.hasNext()){
```

```
        System.out.print(li.next() + " ");}
```

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

```
        Collections.shuffle(ll);
```

```
        // display randomized list
```

```
        li = ll.iterator();
```

```
        System.out.print("List shuffled: ");
```

```
        while(li.hasNext()){
```

```
            System.out.print(li.next() + " ");
```

```
        }
```

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

```
        System.out.println("Minimum: " +  
        Collections.min(ll));
```

```
        System.out.println("Maximum: " +  
        Collections.max(ll));}}
```

This would produce the following result:

List sorted in reverse: 20 8 -8 -20

List shuffled: 20 -20 8 -8

Minimum: -20

Maximum: 20

Benefits of Java Collections Framework

- The Java Collections Framework provides the following benefits:
 - **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
 - **Increases program speed and quality:** The Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.
 - **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.

Contd..

- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Assignment:

- 1) Discuss the difference between List, Set and Queue?
- 2) Write a program to create three list objects l1,l2, and l3. and copy the content of l1 and l2 in to l3
- 3) Create a Map object of minimum 20 employees in ASTU as m1<empname,salary> and
 - 1) find total, minimum, and maximum salary.
 - 2) display name of employee with minimum, maximum and Average salary.
 - 3) Sort and display employee detail by name
- 4) Explain why and when we use List,set and Map with concrete example

Thank You!!

