Collections Framework

java.util package

Introduction to Collections

- **A *collection***
  - is a container
  - is an object that groups multiple elements into a single unit
  - used to store, retrieve, manipulate, and communicate aggregate data.
- **They represent data items that form a natural group**
  - a poker hand (a collection of cards)
  - a mail folder (a collection of letters),
  - a telephone directory (a mapping of names to phone numbers)

What Is a Collections Framework?

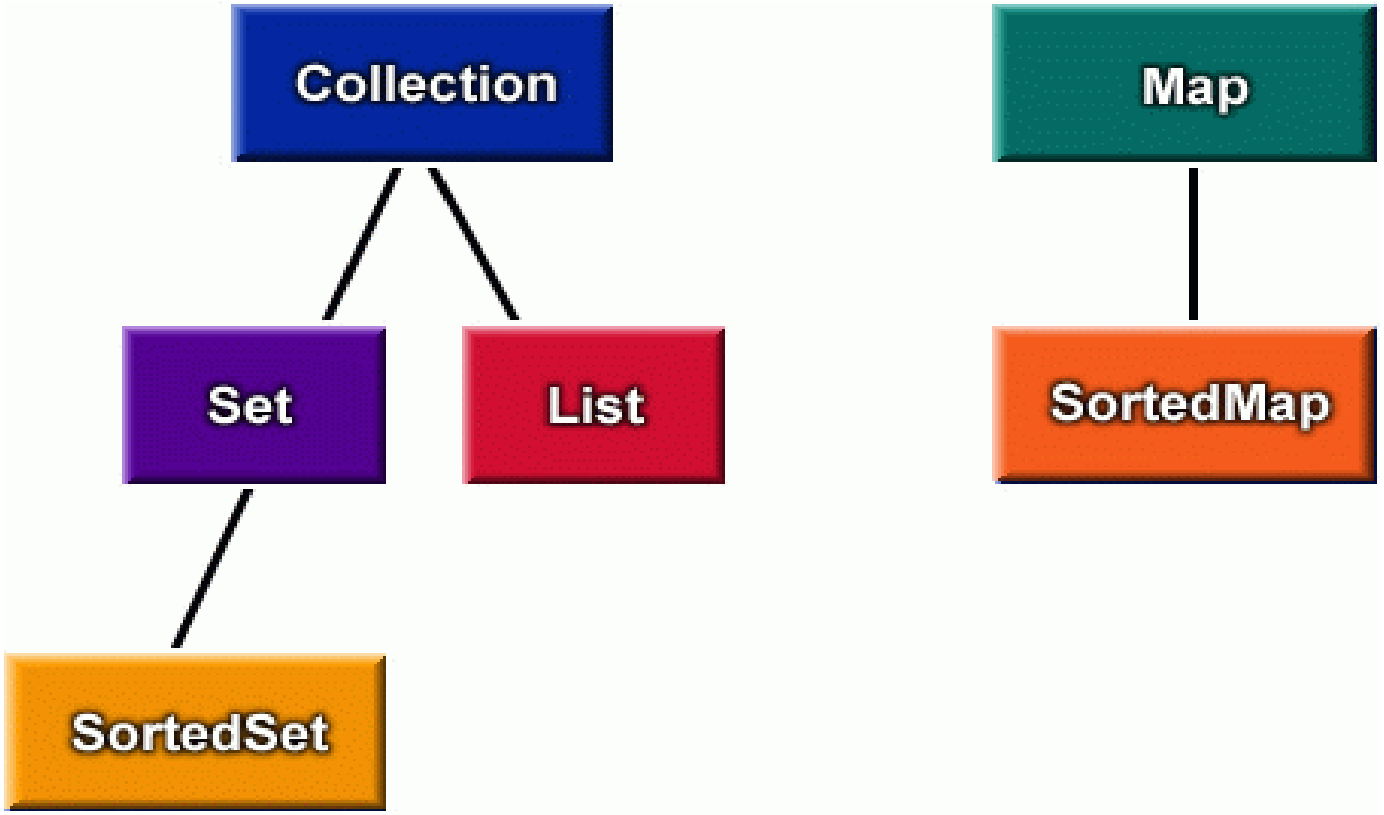
- **A *collections framework* is a unified architecture for representing and manipulating collections**
- **All collections frameworks contain**
  - **Interfaces**
    - These are abstract data types that represent collections
  - **Implementations**
    - These are the concrete implementations of the collection interfaces
    - they are reusable data structures
  - **Algorithms**
    - These are the methods that perform useful computations
    - The algorithms are said to be *polymorphic*

Benefits of the Java Collections Framework

- Reduces programming effort
- Increases program speed and quality
- It allows interoperability among unrelated APIs
- It reduces the effort to learn and use new APIs
- It reduces effort to design new APIs
- It fosters software reuse

core collection interfaces

- Core collection interfaces are the foundation of the Java Collections Framework



The Collection Interface

- **Collection — the root of the collection hierarchy**
- **A collection represents a group of objects known as its *elements***
- **Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered.**
- **The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as *Set* and *List.***

The Collection Interface

```
public interface Collection {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);
    boolean remove(Object element);
     Iterator iterator();

    // Bulk operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c); //optional
    boolean removeAll(Collection c);       //optional
    boolean retainAll(Collection c);       //optional
    void clear();                          //optional

    // Array operations
    Object[]  toArray();
     Object[]  toArray(Object[] a);
}
```

Methods of Collection Framework

Method Summary	
boolean	add(Object o)
boolean	addAll(Collection c)
void	clear()
boolean	contains(Object o)
boolean	containsAll(Collection c)
boolean	equals(Object o)
int	hashCode()
boolean	isEmpty()
Iterator	iterator()
boolean	remove(Object o)
boolean	removeAll(Collection c)
boolean	retainAll(Collection c)
int	size()
Object[]	toArray()
Object[]	toArray(Object[] a)

Traversing Collections

- **To traverse the collection we need to use Iterators**
  - An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired.
- **You get an Iterator for a collection by calling its `iterator` method.**
- **The following is the Iterator interface.**

```
public interface Iterator {
boolean hasNext();
Object next();
void remove(); //optional
}
```

The 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**
- **Adds the restriction that duplicate elements are prohibited**

The Set interface

```
public interface Set extends Collection {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);         //optional
    boolean remove(Object element); //optional
    Iterator iterator();

    // Bulk operations
    boolean containsAll(Collection c);
    boolean addAll(Collection> c); //optional
    boolean removeAll(Collection c);        //optional
    boolean retainAll(Collection c);        //optional
    void clear();                           //optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object[] a);
}
```

Implementations of Set interface

- **The Java platform contains three general-purpose Set implementations**
- **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)

HashSet : example

```
import java.util.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
            if (!s.add(a))
                System.out.println("Duplicate detected: " + a);

        System.out.println(s.size() + " distinct words: " + s);
    }
}
```

Now run the program.
java FindDups i came i saw i left

The following output is produced.
Duplicate detected: i
Duplicate detected: i
4 distinct words: [i, left, saw, came]

The *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
  - **Search**
    - searches for a specified object in the list and returns its numerical position
  - **Iteration**
    - extends Iterator semantics to take advantage of the list's sequential nature
  - **Range-view**
    - performs arbitrary *range operations* on the list.

The List interface

```
public interface List extends Collection {
    // Positional access
    Object get(int index);
    Object set(int index, Object element);    //optional
    boolean add(E element);         //optional
    void add(int index, Object element); //optional
    Object remove(int index);           //optional
    boolean addAll(int index,Collection c); //optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Range-view
    List subList(int from, int to);
}
```

List implementations

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

- **Also, Vector has been reengineered to implement List.**

LinkedList :Example

```
import java.util.*;
 class LinkedListDemo {
 public static void main(String args[]) {
 LinkedList list = new LinkedList();
 list.add(new Integer(1));
 list.add(new Integer(2));
 list.add(new Integer(3));
 list.add(new Integer(1));
 System.out.println(list+", size = "+list.size());
 list.addFirst(new Integer(0));
 list.addLast(new Integer(4));
 System.out.println(list);
 System.out.println(list.getFirst() + ", " + list.getLast());
```

LinkedList :Example

```
//continuation...
 System.out.println(list.get(2)+", "+list.get(3));
 list.removeFirst();
 list.removeLast();
 System.out.println(list);
 list.remove(new Integer(1));
 System.out.println(list);
 list.remove(2);
 System.out.println(list);
 list.set(1, "one");
 System.out.println(list);
 }
}
```

ArrayList : Example

- Definition:
  - **Resizable version an ordinary array**
  - **Implements the *List* interface**

```
import java.util.*;
class ArrayListDemo {
public static void main(String args[]) {
ArrayList al = new ArrayList(2);
System.out.println(al+", size = "+al.size());
al.add("R");
//continued...
```

ArrayList : Example

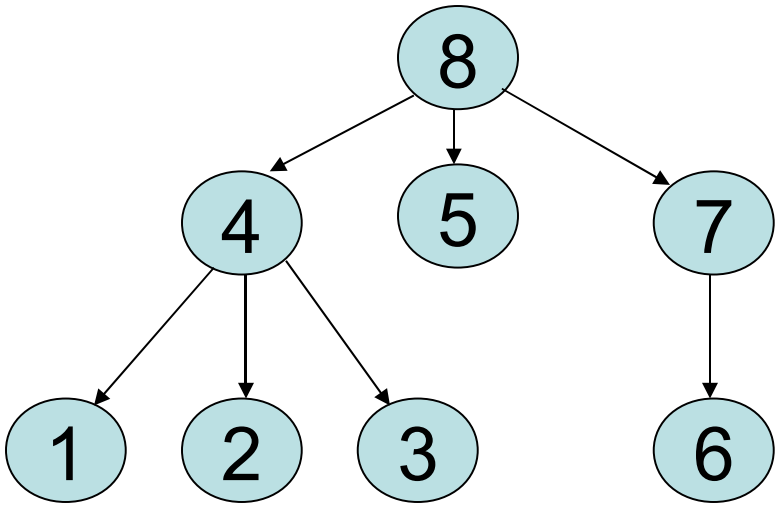
```
    al.add("U");
   al.add("O");
   System.out.println(al+", size = "+al.size());
   al.remove("U");
   System.out.println(al+", size = "+al.size());

   ListIterator li = al.listIterator();
     while (li.hasNext())
    System.out.println(li.next());
    Object a[] = al.toArray();
            for (int i=0; i<a.length; i++)
           System.out.println(a[i]);
     }
}
```

The SortedSet interface

- **A SortedSet is a Set that maintains its elements in ascending order, sorted according to the elements' natural ordering**
- **In addition to the normal Set operations, the SortedSet interface provides operations for the following:**
  - Range view
    - allows arbitrary range operations on the sorted set
  - Endpoints
    - returns the first or last element in the sorted set
  - Comparator access
    - returns the Comparator, if any, used to sort the set

TreeSet

- Definition:
  - Implementation of the *Set* interface that uses a tree
  - Tree
  - Ensures that the sorted set will be arranged in ascending order
- Tree representation



TreeSet : Example

```
import java.util.*;
class TSDemo{
public static void main( String[] args){
TreeSet ts = new TreeSet();
ts.add("Shantanu");
ts.add("Chandramouli");
ts.add("Arun");//new Integer(5));
ts.add("Pavan");//new Double(6.6));
ts.add("Sowjanya");
System.out.println(ts);

Iterator itr = hs.iterator();
    while(itr.hasNext()){
            System.out.println(itr.next());
    }
  }
};
```

The Map Interface

- **A Map is an object that maps keys to values**
- **A map cannot contain duplicate keys**
  - Each key can map to at most one value.
- **It models the mathematical *function* abstraction**

key (object)	value (object)
Andhra Pradesh →	Hyderabad
Madhya Pradesh →	Bhopal
West Bengal →	Kolkata

The Map Interface

```
public interface Map {
    // Basic Operations
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk Operations
    void putAll(Map t);
    void clear();

    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interface for entrySet elements
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

Implementations of Map

- **HashMap**
  - which stores its entries in a hash table, is the best-performing implementation.

- **Hashtable has been retrofitted to implement Map**
- **Other implementations**
  - Attributes, AuthProvider, ConcurrentHashMap, EnumMap, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

HashTable:example

```
import java.util.*;

class HashTableDemo2
{
	public static void main(String[] args)
	{
	Hashtable ht = new Hashtable();
	ht.put("ap", "hyderabad");
	ht.put("kerala","tiruvananthapuram");
	ht.put("tamilnadu", "chennai");
	ht.put("Jaharkhand","Ranchi");

	System.out.println(ht);

	Set s=ht.keySet();
	Iterator i= s.iterator();
	while (i.hasNext()){
		Object o = i.next();
	String s1= (String)o;
	System.out.println("The cap. of "+s1+" is "+ht.get(s1));
	}
}
};
```

HashMap :Example

```
import java.util.*;

class HashMapDemo
{
    public static void main(String[] args)
    {
        HashMap hm = new HashMap();
        //TreeMap hm = new TreeMap();
        hm.put("shantanu", new Double(5000));
        hm.put("Sajid", new Double(3000.50));
        hm.put("Obul Reddy", new Double(500.90));
        hm.put("Manjula", new Double(4000.50));
        hm.put("Rajender",new Double(400.86));

//continued…
```

HashMap : Example

```
Set s = hm.entrySet();
        Iterator itr = s.iterator();
        while (itr.hasNext())
        {
                Map.Entry m =(Map.Entry)itr.next();
                System.out.print( m.getKey()+" : ");
                System.out.println( m.getValue());

        }

        double balance = ((Double)hm.get("shantanu")).doubleValue();
        hm.put("shantanu",new Double(balance+3000));
        System.out.print("shantanu's new Balance is..");
        System.out.println(hm.get("shantanu"));
                }
}
```

SortedMap interface

- **A SortedMap is a Map that maintains its entries in ascending order**
  - sorted according to the keys' *natural order*, or according to a Comparator provided at SortedMap creation time
- **Additional methods of SortedMap**
  - Range-view:
    - Performs arbitrary *range operations* on the sorted map.
  - Endpoints:
    - Returns the first or last key in the sorted map.
  - Comparator access:
    - Returns the Comparator used to sort the map (if any)

SortedMap interface

```
public interface SortedMap extends Map {
    Comparator comparator();

    SortedMap subMap(Object fromKey, Object toKey);
    SortedMap headMap(Object toKey);
    SortedMap tailMap(Object fromKey);

    Object firstKey();
    Object lastKey();
}
```

Implementations of SortedMap

- **TreeMap**
  - Red-Black tree based implementation of the SortedMap interface.
  - This class guarantees that the map will be in ascending key order, sorted according to the *natural order* for the key's class
  - or by the comparator provided at creation time, depending on which constructor is used.
- **This implementation is not synchronized**
  - If multiple threads access a map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally.

TreeMap :Example

```
import java.util.*;

class TreeMapDemo
{
    public static void main(String[] args)
    {
        TreeMap hm = new TreeMap();
        hm.put("shantanu", new Double(5000));
        hm.put("Sajid", new Double(3000.50));
        hm.put("Obul Reddy", new Double(500.90));
        hm.put("Manjula", new Double(4000.50));
        hm.put("Rajender",new Double(400.86));

//continued…
```

TreeMap : Example

```
Set s = hm.entrySet();
        Iterator itr = s.iterator();
        while (itr.hasNext())
        {
                Map.Entry m =(Map.Entry)itr.next();
                System.out.print( m.getKey()+" : ");
                System.out.println( m.getValue());

        }

        double balance = ((Double)hm.get("shantanu")).doubleValue();
        hm.put("shantanu",new Double(balance+3000));
        System.out.print("shantanu's new Balance is..");
        System.out.println(hm.get("shantanu"));
                }
}
```

Thank You