**UNIT – IV**

**Java Collections**

**Collections Overview**

The Java Collections Framework standardizes the way in which groups of objects are handled by your programs.

The Collections Framework was designed to meet several goals.

**First**, the framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.

**Second**, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.

**Third**, extending and/or adapting a collection had to be easy. Toward this end, the entire Collections Framework is built upon a set of standard interfaces.

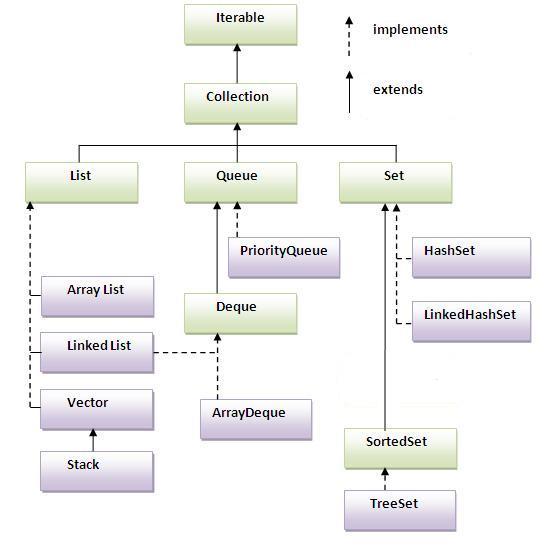
*Algorithms* are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the **Collections** class. Thus, they are available for all collections. Each collection class need not implement its own versions. The algorithms provide a standard means of manipulating collections.

Another item closely associated with the Collections Framework is the **Iterator** interface. An *iterator* offers a general-purpose, standardized way of accessing the elements within a collection, one at a time. Thus, an iterator provides a means of *enumerating the contents of* *a collection.* Because each collection implements **Iterator**, the elements of any collection class can be accessed through the methods defined by **Iterator**. Thus, with only small changes, the code that cycles through a set can also be used to cycle through a list, for example. Iterator interface provides the facility of iterating the elements in forward direction only. There are only three methods in the Iterator interface. They are:

1. **public boolean hasNext()** it returns true if iterator has more elements.
2. **public object next()** it returns the element and moves the cursor pointer to the next element.
3. **public void remove()** it removes the last elements returned by the iterator. It is rarely used.

### Hierarchy of Collection Framework

Let us see the hierarchy of collection framework. The **java.util** package contains all the classes and interfaces for Collection framework.



**Collection Interfaces:**

**The Collection interface:**

The Collection interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have. These methods are summarized in the following.

**boolean add(Object obj)**

Adds obj to the invoking collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.

**boolean addAll(Collection c)**

Adds all the elements of c to the invoking collection. Returns true if the operation succeeded (i.e., the elements were added). Otherwise, returns false.

**Iterator iterator( )**

Returns an iterator for the invoking collection.

**boolean remove(Object obj)**

Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.

**boolean removeAll(Collection c)**

Removes all elements of c from the invoking collection. Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.

**int size( )**

Returns the number of elements held in the invoking collection.

**Object[ ] toArray( )**

Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.

**The List interface:**

The List interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.

* Elements can be inserted or accessed by their position in the list, using a zero-based index.
* A list may contain duplicate elements.
* In addition to the methods defined by **Collection**, List defines some of its own, which are summarized in the following below Table.
* Several of the list methods will throw an UnsupportedOperationException if the collection cannot be modified, and a ClassCastException is generated when one object is incompatible with another.

**Object get(int index)**

Returns the object stored at the specified index within the invoking collection.

**int indexOf(Object obj)**

Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.

**int lastIndexOf(Object obj)**

Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.

**ListIterator listIterator( )**

Returns an iterator to the start of the invoking list.

**ListIterator listIterator(int index)**

Returns an iterator to the invoking list that begins at the specified index.

**Object set(int index, Object obj)**

Assigns obj to the location specified by index within the invoking list.

**List subList(int start, int end)**

Returns a list that includes elements from start to end.1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

**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.

**add( )**

Adds an object to the collection

**clear( )**

Removes all objects from the collection

**contains( )**

Returns true if a specified object is an element within the collection

**remove( )**

Removes a specified object from the collection

## Example:

Set have its implementation in various classes like HashSet, TreeSet, LinkedHashSet. Following is the example to explain Set functionality:

import java.util.\*;

public class SetDemo {

public static void main(String args[]) {

int count[] = {34, 22,10,60,30,22};

Set<Integer> set = new HashSet<Integer>();

try{

for(int i = 0; i<5; i++){

set.add(count[i]);

}

System.out.println(set);

TreeSet sortedSet = new TreeSet<Integer>(set);

System.out.println("The sorted list is:");

System.out.println(sortedSet);

System.out.println("The First element of the set is: "+

(Integer)sortedSet.first());

System.out.println("The last element of the set is: "+

(Integer)sortedSet.last());

}

catch(Exception e){}

}

}

**The Map interface:**

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

* Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.
* Several methods throw a NoSuchElementException when no items exist in the invoking map.
* A ClassCastException is thrown when an object is incompatible with the elements in a map.
* A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.

**void clear( )**

Removes all key/value pairs from the invoking map.

**boolean containsKey(Object k)**

Returns true if the invoking map contains k as a key. Otherwise, returns false.

**boolean containsValue(Object v)**

Returns true if the map contains v as a value. Otherwise, returns false

**Object get(Object k)** R

eturns the value associated with the key k.

**int hashCode( )**

Returns the hash code for the invoking map.

**Set keySet( )**

Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.

**Object put(Object k, Object v)**

Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.

**void putAll(Map m)**

Puts all the entries from m into this map.

**Object remove(Object k)**

Removes the entry whose key equals k.

**int size( )**

Returns the number of key/value pairs in the map.

**Collection values( )**

Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Example:

import java.util.\*;

public class CollectionsDemo {

public static void main(String[] args) {

Map m1 = new HashMap();

m1.put("Zara", "8");

m1.put("Mahnaz", "31");

m1.put("Ayan", "12");

m1.put("Daisy", "14");

System.out.println();

System.out.println(" Map Elements");

System.out.print("\t" + m1);

}

}

**The Queue Interface**

The **Queue** interface extends **Collection** and declares the behavior of a queue, which is often a first-in, first-out list. However, there are types of queues in which the ordering is based upon other criteria. **Queue** is a generic interface that has this declaration:

interface Queue<E>



Since Queue is an interface you need to instantiate a concrete implementation of the interface in order to use it. You can choose between the following Queue implementations in the Java Collections API:

* java.util.LinkedList
* java.util.PriorityQueue

import java.util.\*;

public class QueueDemo {

static String newLine = System.getProperty("line.separator");

public static void main(String[] args) {

System.out.println(newLine + "Queue in Java" + newLine);

System.out.println("-----------------------" + newLine);

System.out.println("Adding items to the Queue" + newLine);

//Creating queue would require you to create instannce of LinkedList and assign

//it to Queue

//Object. You cannot create an instance of Queue as it is abstract

Queue queue = new LinkedList();

//you add elements to queue using add method

queue.add("Java");

queue.add(".NET");

queue.add("Javascript");

queue.add("HTML5");

queue.add("Hadoop");

System.out.println(newLine + "Items in the queue..." + queue + newLine);

//You remove element from the queue using .remove method

//This would remove the first element added to the queue, here Java

System.out.println("remove element: " + queue.remove() + newLine);

//.element() returns the current element in the queue, here when "java" is removed

//the next most top element is .NET, so .NET would be printed.

System.out.println("retrieve element: " + queue.element() + newLine);

//.poll() method retrieves and removes the head of this queue

//or return null if this queue is empty. Here .NET would be printed and then would

//be removed

//from the queue

System.out.println("remove and retrieve element, null if empty: " + queue.poll() +

newLine);

//.peek() just returns the current element in the queue, null if empty

//Here it will print Javascript as .NET is removed above

System.out.println("retrieve element, null is empty " + queue.peek() + newLine);

}

}

**Collection Classes:**

**The ArrayList Class:**

The **ArrayList** class extends **AbstractList** and implements the **List** interface. **ArrayList** is a generic class that has this declaration:

class ArrayList<E>

Here, **E** specifies the type of objects that the list will hold.

**ArrayList** supports dynamic arrays that can grow as needed. In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, then Collections Framework defines **ArrayList**. In essence, an **ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

**ArrayList** has the constructors shown here:

ArrayList( )

ArrayList(Collection<? extends E> *c*)

ArrayList(int *capacity*)

import java.util.\*;

class ArrayListDemo {

public static void main(String args[]) {

// Create an array list.

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

System.out.println("Initial size of al: " +

al.size());

// Add elements to the array list.

al.add("C");

al.add("A");

al.add("E");

al.add("B");

al.add("D");

al.add("F");

al.add(1, "A2");

System.out.println("Size of al after additions: " +

al.size());

// Display the array list.

System.out.println("Contents of al: " + al);

// Remove elements from the array list.

al.remove("F");

al.remove(2);

System.out.println("Size of al after deletions: " +

al.size());

System.out.println("Contents of al: " + al);

}

}

Convert an ArrayList into an array.

import java.util.\*;

class ArrayListToArray {

public static void main(String args[]) {

// Create an array list.

ArrayList<Integer> al = new ArrayList<Integer>();

// Add elements to the array list.

al.add(1);

al.add(2);

al.add(3);

al.add(4);

System.out.println("Contents of al: " + al);

// Get the array.

Integer ia[] = new Integer[al.size()];

ia = al.toArray(ia);

int sum = 0;

// Sum the array.

for(int i : ia) sum += i;

System.out.println("Sum is: " + sum);

}

}

**Linked list:**

A **linked list** is a data structure used for collecting a sequence of objects that allows efficient addition and removal of elements in the middle of the sequence.

Rather than storing the values in an array, a linked list uses a sequence of *nodes*. Each node stores a value and a reference to the next node in the sequence . When you insert a new node into a linked list, only the neighbouring node references need to be updated. The same is true when you remove a node. What’s the catch? Linked lists allow speedy insertion and removal, but element access can be slow.

|  |
| --- |
|  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Stack:**

**Stack** is a subclass of **Vector** that implements a standard last-in, first-out stack. **Stack** only defines the default constructor, which creates an empty stack.

class Stack<E>

Here, **E** specifies the type of element stored in the stack.



import java.util.\*;

class StackDemo {

static void showpush(Stack<Integer> st, int a) {

st.push(a);

System.out.println("push(" + a + ")");

System.out.println("stack: " + st);

}

static void showpop(Stack<Integer> st) {

System.out.print("pop -> ");

Integer a = st.pop();

System.out.println(a);

System.out.println("stack: " + st);

}

public static void main(String args[]) {

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

System.out.println("stack: " + st);

showpush(st, 42);

showpush(st, 66);

showpush(st, 99);

showpop(st);

showpop(st);

showpop(st);

try {

showpop(st);

} catch (EmptyStackException e) {

System.out.println("empty stack");

}

}

}