

Collection is basically any group of individual objects.

Framework = Classes + Interfaces

which provides ready-made structure like we can just use their functionality by just applying some methods on them.

Frameworks of Collection Framework---

1. Iterable Interface --

root of Collection framework, Collection Interface extends this interface mtlb indirectly jo bi collection ko implement krta h usme iterable interface bi aata h. main functionality of iterable interface is to provide iterator for the collections.

that's why this interface **java.util** contains only one abstract method which is the **iterator**, i.e., **Iterator** iterator ();

Collections Framework

2. Collection Interface --

extends Iterable interface, all classes of Collection Framework implements collection interface.

This interface contains all the basic methods for every collection like adding data, removing, etc.

List, Queue, Set are the child interfaces of collection interface.

Introduction

- Support wide range of classes and interfaces with broad range of functionality
- Our focus
 - ArrayList
 - Map
 - HashMap
 - Iterator

The ArrayList Class

- Extends `AbstractList` and implements the `List` interface.
- It is a generic class: `class ArrayList<E>`, where, `E` specifies the type of objects.
- Supports dynamic arrays
- When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

Example : ArrayList

```
import java.util.*;
class ArrayListDemo {
    public static void main(String args[]) {
        ArrayList<String> al = new ArrayList<String>(); // Create an array list.
        System.out.println("Initial size of al: " +al.size());

        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());

        System.out.println("Contents of al: " + al);
        al.remove("F");
        al.remove(2);
        System.out.println("Size of al after deletions: " +al.size());
        System.out.println("Contents of al: " + al);
    }
}
```

ArrayList methods

- We can still increase the capacity of an ArrayList object manually by calling `ensureCapacity()`.
 - `void ensureCapacity(int cap) // cap is the new capacity.`
- Conversely, we can reduce the size of the array that underlies an ArrayList object so that it is precisely as large as the number of items that it is currently holding, by calling `trimToSize()`,
 - `void trimToSize()`
- Converting an ArrayList to an array is a trivial task.
 - `Object[] toArray()`
 - `<T> T[] toArray(T array[])`

Example: toArray()

```
import java.util.*;
class ArrayListToArray {
    public static void main(String args[]) {
        ArrayList<Integer> al = new ArrayList<Integer>();

        al.add(1);
        al.add(2);
        al.add(3);
        al.add(4);
        System.out.println("Contents of al: " + al);

        Integer ia[] = new Integer[al.size()];
        ia = al.toArray(ia);
        int sum = 0;
        for(int i : ia)
            sum += i;
        System.out.println("Sum is: " + sum);
    }
}
```

Accessing a Collection via an Iterator

- Iterator is an object that implements either the Iterator or the ListIterator interface.
- Enables you to cycle through a collection, obtaining or removing elements.
- ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
- Iterator and ListIterator are generic interfaces
 - `interface Iterator<E>`
 - `interface ListIterator<E>` //E specifies the type of objects being iterated
- **Note:** If you won't be modifying the contents of a collection or obtaining elements in reverse order, then the for-each loop is often a more convenient than is using an iterator.

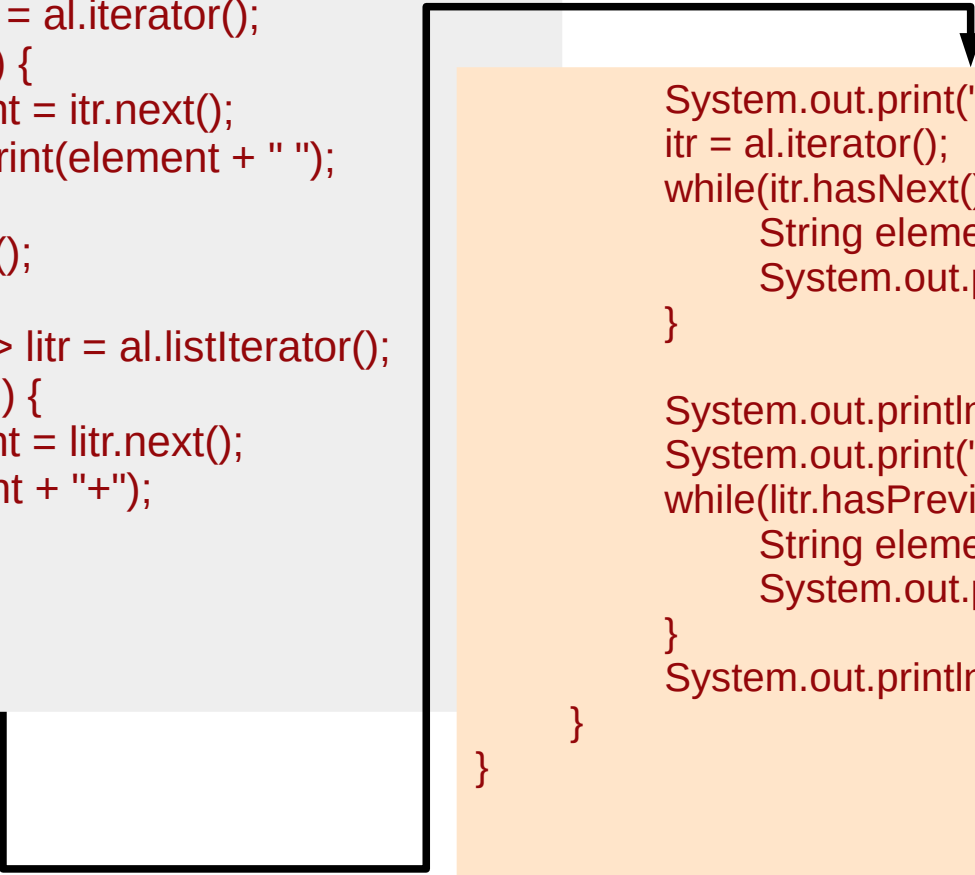
Example :

Iterator

```
import java.util.*;
class IteratorDemo {
    public static void main(String args[]) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
```

```
        System.out.print("Original contents of al: ");
        Iterator<String> itr = al.iterator();
        while(itr.hasNext()) {
            String element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();
```

```
        ListIterator<String> litr = al.listIterator();
        while(litr.hasNext()) {
            String element = litr.next();
            litr.set(element + "+");
        }
```



```
        System.out.print("Modified contents of al: ");
        itr = al.iterator();
        while(itr.hasNext()) {
            String element = itr.next();
            System.out.print(element + " ");
        }
```

```
        System.out.println();
        System.out.print("Modified list backwards: ");
        while(litr.hasPrevious()) {
            String element = litr.previous();
            System.out.print(element + " ");
        }
        System.out.println();
```

```
    }
}
```


Map

- A map is an object that stores associations between keys and values, or key/value pairs.
- Both keys and values are objects.
- The keys must be unique, but the values may be duplicated.
- Some maps can accept a null key and null values, others cannot.
- They don't implement the Iterable interface.
- Can't obtain an iterator to a map. However, we can obtain a collection-view of a map, which does allow the use of either the for loop or an iterator.
 - **interface Map<K, V>**
- Here, K specifies the type of keys, and V specifies the type of values.

Map basic methods

- **V put(K k, V 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.
- **V get(Object k) :**
 - Returns the value associated with the key k.
 - Returns null if the key is not found
- We can obtain a collection-view of a map by using the **entrySet()** method which returns a Set that contains the elements in the map.
- Obtain a collection-view of the keys by using **keySet()** method
- Obtain a collection-view of the values by using **values()** method

The HashMap Class

- The HashMap class extends AbstractMap and implements the Map interface.
- It uses a hash table to store the map.
- This allows the execution time of get() and put() to remain constant even for large sets.
- Declaration : `class HashMap<K, V>`
- Constructors:
 - `HashMap()`
 - `HashMap(Map<? extends K, ? extends V> m)`
 - `HashMap(int capacity)`
 - `HashMap(int capacity, float fillRatio)`

Example : HashMap

```
import java.util.*;
class HashMapDemo {
    public static void main(String args[]) {
        HashMap<String, Double> hm = new HashMap<String, Double>();

        hm.put("John Doe", new Double(3434.34));
        hm.put("Tom Smith", new Double(123.22));
        hm.put("Jane Baker", new Double(1378.00));
        hm.put("Tod Hall", new Double(99.22));
        hm.put("Ralph Smith", new Double(-19.08));

        // Get a set of the entries.
        Set<Map.Entry<String, Double>> set = hm.entrySet();
        // Display the set.
        for(Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();
        // Deposit 1000 into John Doe's account.
        double balance = hm.get("John Doe");
        hm.put("John Doe", balance + 1000);
        System.out.println("John Doe's new balance: " +
            hm.get("John Doe"));
    }
}
```

Further Reading

- All remaining commonly used java.util classes and interfaces and practice demonstration examples.