# OBJECT ORIENTED PROGRAMMING (ICT 2155)

# Collections Framework.

- Hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects.

- Present in **java.util** package.

Framework: an essential supporting structure

## Collections Overview

- Collections Framework standardizes the way in which groups of objects are handled in our programs.

- Collections were not part of the original Java release, but were added by J2SE 1.2.

- Prior to the Collections Framework:

  - ✓ Java provided ad hoc classes such as Dictionary, Vector, Stack, and Properties to store and manipulate groups of objects.

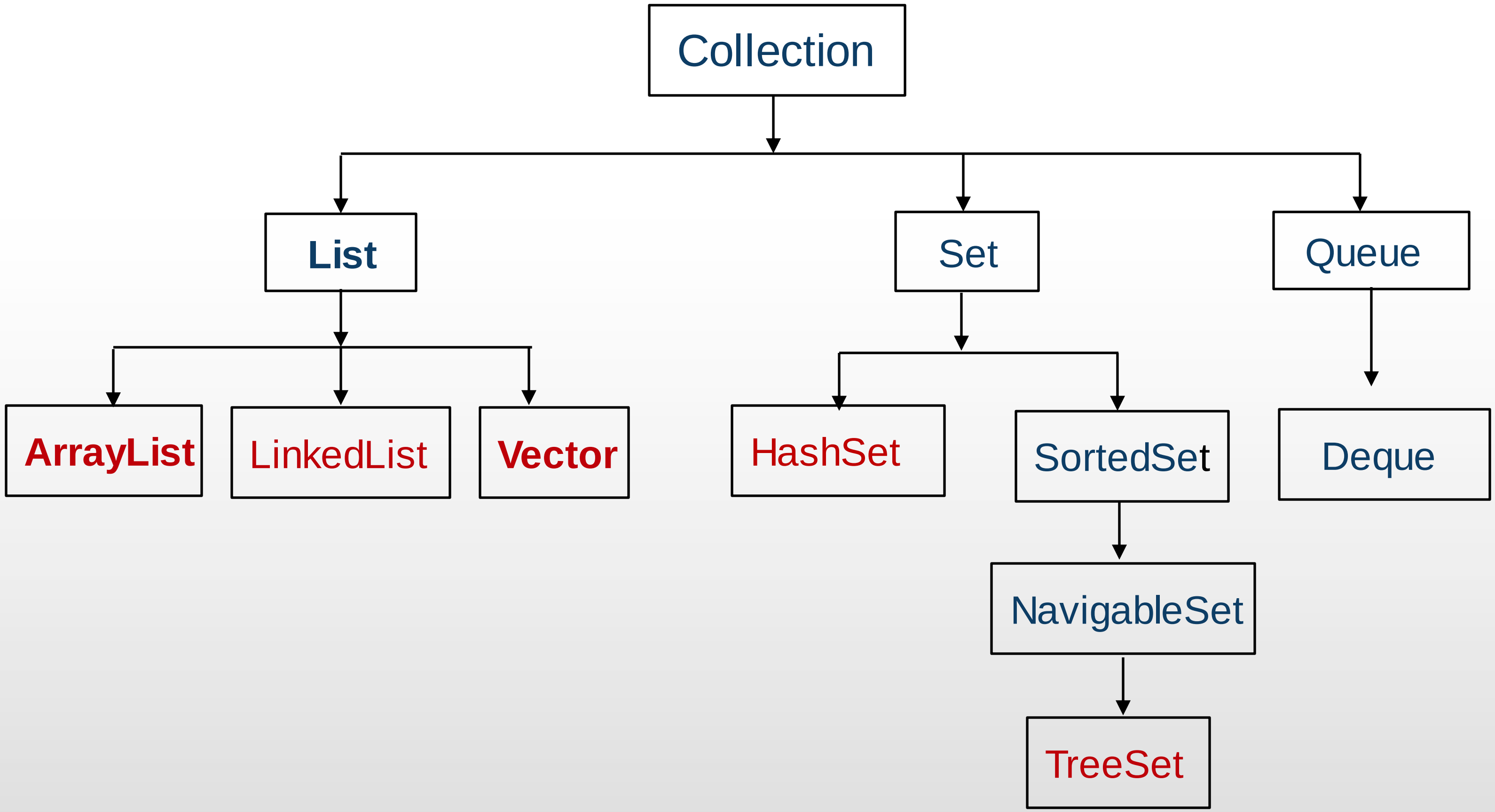
  - ✓ They lacked a central, unifying theme.

# Collections Overview

- Entire Collections Framework is built upon a set of standard interfaces.

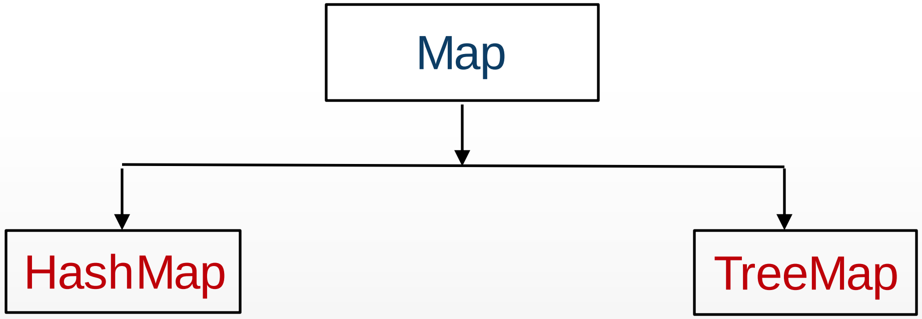
- Each interface type is implemented by one or more classes

- Each class is designed for a specific type of storage.




## Collection Interfaces

The Collections Framework defines several interfaces.


Map
HashMap
TreeMap

# The Collection Interface

The **Collection** interface is the foundation upon which the Collections Framework is built.

It must be implemented by any class that defines a collection.

**Collection** is a generic interface that has this declaration:

interface Collection<E>

Here, **E** specifies the type of objects that the collection will hold.
**Collection** extends the **Iterable** interface.
all collections can be cycled through by use of the for-each style **for** loop.

# The `Collection` Interface

List, Queue and Set are specialized interfaces that inherit from the Collection interface. All share the following commonly used methods

**Table 1 The Methods of the `Collection` Interface**

| | |
|---|---|
| `Collection<String> coll = new ArrayList<String>();` | The `ArrayList` class implements the `Collection` interface. |
| `coll = new TreeSet<String>()` | The `TreeSet` class (Section 15.3) also implements the `Collection` interface. |
| `int n = coll.size();` | Gets the size of the collection. `n` is now `0`. |
| `coll.add("Harry");` `coll.add("Sally");` | Adds elements to the collection. |
| `String s = coll.toString();` | Returns a string with all elements in the collection. `s` is now `"[Harry, Sally]"` |
| `System.out.println(coll);` | Invokes the `toString` method and prints `[Harry, Sally]`. |

| | |
|---|---|
| `coll.remove("Harry");`<br>`boolean b = coll.remove("Tom");` | Removes an element from the collection, returning `false` if the element is not present. `b` is false. |
| `b = coll.contains("Sally");` | Checks whether this collection contains a given element. `b` is now `true`. |
| `for (String s : coll)`<br>`{`<br>`   System.out.println(s);`<br>`}` | You can use the "for each" loop with any collection. This loop prints the elements on separate lines. |
| `Iterator<String> iter = coll.iterator()` | You use an iterator for visiting the elements in the collection (see Section 15.2.3). |

**Collection** declares the core methods that all collections will have.

| Method | Description |
|---|---|
| boolean add(E *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 and the collection does not allow duplicates. |
| boolean addAll(Collection<? extends E> *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**. |
| void clear( ) | Removes all elements from the invoking collection. |
| boolean contains(Object *obj*) | Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**. |
| boolean containsAll(Collection<?> *c*) | Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**. |
| boolean equals(Object *obj*) | Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**. |
| int hashCode( ) | Returns the hash code for the invoking collection. |
| boolean isEmpty( ) | Returns **true** if the invoking collection is empty. Otherwise, returns **false**. |
| Iterator<E> 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**. |
| boolean retainAll(Collection<?> *c*) | Removes all elements from the invoking collection except those in *c*. 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. |

# 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.

**List** is a generic interface that has this declaration:

interface List<E>

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

In addition to the methods defined by **Collection**, **List** defines some of its own,

| Method | Description |
|---|---|
| void add(int *index*, E *obj*) | Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. |
| boolean addAll(int *index*, Collection<? extends E> *c*) | Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise. |
| E 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<E> listIterator( ) | Returns an iterator to the start of the invoking list. |
| ListIterator<E> listIterator(int *index*) | Returns an iterator to the invoking list that begins at the specified index. |
| E remove(int *index*) | Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one. |
| E set(int *index*, E *obj*) | Assigns *obj* to the location specified by *index* within the invoking list. |
| List<E> 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. |

**TABLE 17-2** The Methods Defined by **List**

# List

- Ordered Lists

  - `ArrayList`
    - Stores a list of items in a dynamically sized array
  - `LinkedList`
    - Allows speedy insertion and removal of items from the list

A **list** is a collection that maintains the order of its elements.

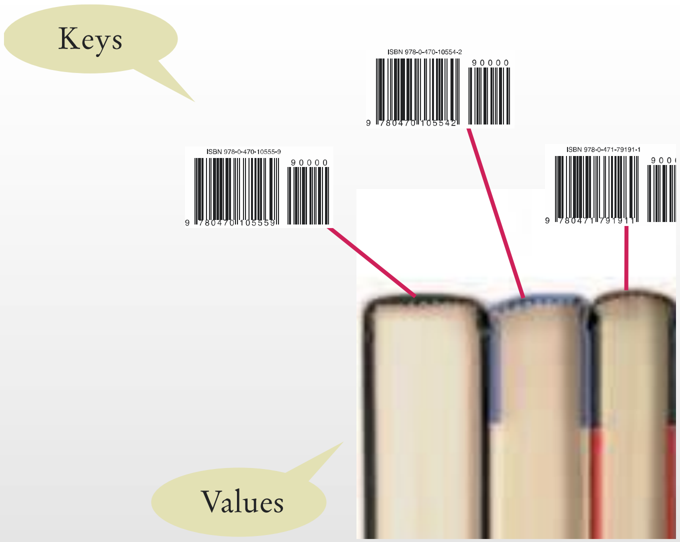
# Set

- Unordered Sets

  - `HashSet`
    - Uses hash tables to speed up finding, adding, and removing elements
  - `TreeSet`
    - Uses a binary tree to speed up finding, adding, and removing elements

# Maps

- A map stores keys, values, and the associations between them
  - Example:
  - Barcode keys and books




A map keeps associations between key and value objects.

- Keys
  - Provides an easy way to represent an object (such as a numeric bar code)
- Values
  - The actual object that is associated with the key

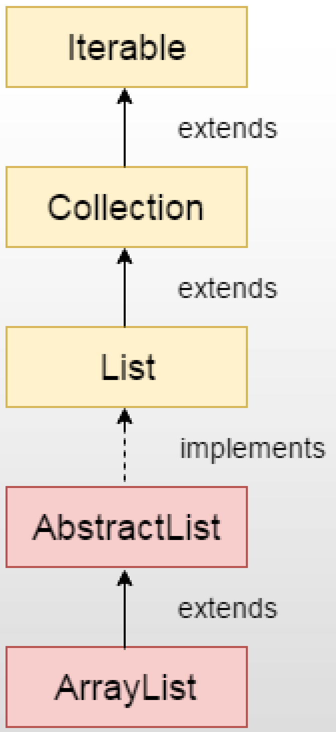
# Array List

- ArrayList is a part of collection framework.

- Present in java.util package.

- Java ArrayList class uses a dynamic array for storing the elements.

- It inherits AbstractList class and implements List interface.



# Array List

- ArrayList slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

- ArrayList is initialized by a size, however the size can increase if collection grows or shrunk if objects are removed from the collection.

- Java ArrayList allows us to randomly access the list.

- ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.

The important points about Java ArrayList class are:

- ArrayList class can contain duplicate elements.
- ArrayList class maintains insertion order.
- ArrayList class is non synchronized.
- In ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

```
ArrayList<type>  arr = new ArrayList<type>();

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

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

ArrayList<Employee>  arr = new ArrayList<Employee>();
```

```
int n = 5; // size of ArrayList

ArrayList<Integer> arrli = new ArrayList<Integer>(n);

// Appending the new element at the end of the list
for (int i=1; i<=n; i++)
    arrli.add(i);

for (int i=0; i<arrli.size(); i++)
    System.out.print(arrli.get(i)+" ");
```

OUTPUT : 1 2 3 4 5

```
int n = 5; // size of ArrayList

ArrayList<Integer> arrli = new ArrayList<Integer>(n);

// Appending the new element at the end of the list
for (int i=1; i<=n; i++)
    arrli.add(i);

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

OUTPUT : [1 2 3 4 5]

```
ArrayList al = new ArrayList();
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);
```

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

```
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
 System.out.println(itr.next());
}
```

# Iterator

Used to cycle through the elements in a collection.

For example: to display each element.

**Iterator** enables us to cycle through a collection, obtaining or removing elements.

**Iterator** is a generic interfaces which are declared as shown here:

interface Iterator<E>

Here, **E** specifies the type of objects being iterated.

# Iterator

Iterator<String> it = al.iterator();

| Method | Description |
|---|---|
| boolean hasNext( ) | Returns **true** if there are more elements. Otherwise, returns **false.** |
| E next( ) | Returns the next element. Throws **NoSuchElementException** if there is not a next element. |
| void remove( ) | Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( ).** |

# Iterators and Loops

Iterator<String> iterator = al.iterator();

- Iterators are often used in while and "for-each" loops
  - `hasNext` returns true if there is a next element
  - `next` returns a reference to the value of the next element

```
while (iterator.hasNext())
{
   String name = iterator.next();
   // Do something with name
}
```

```
for (String name : employeeNames)
{
   // Do something with name
}
```

Note : in for each loop iterator is used 'behind the scenes'

- To increase the capacity of an **ArrayList** object manually.
- By increasing its capacity once, at the start, we can prevent several reallocations later.
- Reallocations are costly in terms of time, preventing unnecessary ones improves performance.

void ensureCapacity(int *cap*)

Here, *cap* is the new capacity.

If we want to 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 using **trimToSize( )** method.

```
void trimToSize( )
```

## Obtaining an Array from an ArrayList

to obtain an actual array that contains the contents of the list.

**toArray( )** is defined by **Collection**.

Reasons to convert a collection into an array:

- To obtain faster processing times for certain operations
- To pass an array to a method that is not overloaded to accept a collection
- To integrate collection-based code with legacy code that does not understand collections

```
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()];
al.toArray(ia);


int sum = 0;
for(int i : ia)
 sum += i;
System.out.println("Sum is: " + sum);
```

```
import java.util.*;
class Student
{
		int regno;
		String name;

		Student(int a, String b)
		{
			regno = a;
			name = b;
		}
		void disp()
		{
			System.out.println("Regno : "+regno+"   name : "+name);
		}
}
```

```
class ArrayListStud
{
        public static void main(String args[])
        {
                ArrayList<Student> a = new ArrayList<Student>();
                a.add(new Student(1,"anil"));
                a.add(new Student(2,"sunil"));
                a.add(new Student(3,"rahul"));
                a.add(new Student(4,"sachin"));
                a.add(new Student(5,"kiran"));

                for(Student s : a)
                s.disp();
        }
}
```

# Vector

**Vector** implements a dynamic array.

It is similar to **ArrayList**, but with two differences:

- **Vector** is synchronized, and
- it contains many legacy methods that are not part of the Collections

# Vector

With the advent of collections, **Vector** was reengineered to extend **AbstractList** and to implement the **List** interface.

With the release of JDK 5, it was retrofitted for generics and reengineered to implement **Iterable**.

This means that **Vector** is fully compatible with collections, and a **Vector** can have its contents iterated by the enhanced **for** loop.