T. Y. B. Sc.
COMPUTER SCIENCE
SEMESTER-VI

NEW SYLLABUS
CBCS PATTERN

OBJECT ORIENTED PROGRAMMING USING JAVA-II

Dr. Ms. MANISHA BHARAMBE
Ms. MANISHA GADEKAR

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

Syllabus ...

1. Collections (6 Lectures)

- Introduction to the Collection Framework
- List - ArrayList, LinkedList
- Set - HashSet, TreeSet,
- Map - HashMap and TreeMap
- Interfaces such as Comparator, Iterator, ListIterator, Enumeration

2. Multithreading (6 Lectures)

- What are Threads?
- Life cycle of Thread
- Creating Threads:
  - Thread Class
  - Runnable Interface
- Thread Priorities
- Running Multiple Threads
- Synchronization and Interthread Communication

3. Database Programming (6 Lectures)

- The Design of jdbc
- Types of Drivers
- Executing SQL Statements, Query Execution
- Scrollable and Updatable Resultset

4. Servlets and JSP (12 Lectures)

- Introduction to Servlet and Hierarchy of Servlet
- Life Cycle of Servlet
- Handing Get and Post Request (HTTP)
- Handling Data from HTML to Servlet
- Retrieving Data from Database to Servlet
- Session Tracking:
  - User Authorization
  - URL Rewriting
  - Hidden Form Fields
  - Cookies and HttpSession
- Introduction to JSP, Life Cycle of JSP
- Implicit Objects

- Scripting Elements:
  - Declarations
  - Expressions
  - Scriplets
  - Comments
- JSP Directives - Page Directive, Include Directive
- Mixing Scriplets and HTML
- JSP Actions:
  - jsp:forward
  - jsp:include
  - jsp:useBean
  - jsp:setProperty
  - jsp:getProperty

5. Spring Framework

(6 Lecture

- Introduction of Spring Framework
- Spring Modules / Architecture
- Spring Applications
- Spring MVC
- Spring MVC Forms, Validation

■■■

CHAPTER 1

Collections

Objectives...

- To understand Collections and Collection Framework
- To study Collection Interfaces and Classes
- To learn Set and List Interfaces
- To Study Map and Comparator Interfaces

1.0 INTRODUCTION

- The Collection framework which is contained in the java.util package is one of Java's most powerful systems.
- The Collection framework defines a set of interfaces and their implementations to manipulate Collections, which serve as a container for a group of objects such as a collection of mails.
- A Collection is an object that holds other objects. A Collection is a group of objects. In Java, these objects are called elements of the Collection.
- Java Collection framework is a collection of interfaces and classes used to storing and processing a group of individual objects as a single unit.
- The Java Collection framework holds several classes that provide a large number of methods to store and process a group of objects.
- The Collections framework was designed to meet following goals:
  1. The Collection framework had to extend and/or adapt a collection easily.
  2. The framework had to be high-performance.
  3. The framework had to allow different types of Collections to work in a similar manner and with a high degree of interoperability.

1.1 OVERVIEW TO COLLECTION FRAMEWORK [April 17, Oct. 18]

- A Collection, as its name implied, is simply an object that holds a collection (or a group or a container) of objects. Each item in a collection is called an element.

- In short, Collection is simply an object that groups multiple elements into a single un All the operations that we perform on a data such as searching, sorting, insertio manipulation, deletion etc. can be performed by Java Collections.
- The Java Collections framework is a collection of interfaces and classes which helps storing and processing the data efficiently.
- Java Collection means a single unit of objects. A collection represents a group objects, known as its elements.
- A collections framework is a unified architecture for representing and manipulatin Collections.
- All Collections frameworks has:
  1. **Interfaces:** Interfaces are abstract data types that represent collections. The Collection interface is available inside the java.util package. The Collection interface (java.util.Collection) root interface in the Collections hierarchy.
  2. **Implementations i.e., Classes:** Java provides core implementation classes for Collections. The Java Collection framework holds several classes that provide a large number of methods to store and process a group of objects. We can use them to create different types of collections in our program.
  3. **Algorithms:** Algorithms are the useful methods to provide some common functionality such as searching and sorting.

1.1.1 Benefits of Collection Framework

- The Java Collections Framework provides the following benefits:
  1. **Reduces programming effort:** It comes with almost all common types of collections and useful methods to iterate and manipulate the data. So we can concentrate more on business logic rather than designing our collection APIs.
  2. **Increases program speed and quality:** Using core collection classes that are well tested increases our program quality rather than using any home developed data structure. Because we are freed from the work of writing our own data structures we will have more time to devote to improving program's quality and performance.
  3. **Allows interoperability among unrelated APIs:** The collection interfaces are the dialect by which APIs pass collections back and forth.
  4. **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 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.

5. **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.
6. **Fosters software reuse:** Like any New data structures that conform to the standard collection interfaces are also nature reusable.

1.1.2 Collections Framework Hierarchy

[April 18, 19]

- Fig. 1.1 shows Collections Framework hierarchy.

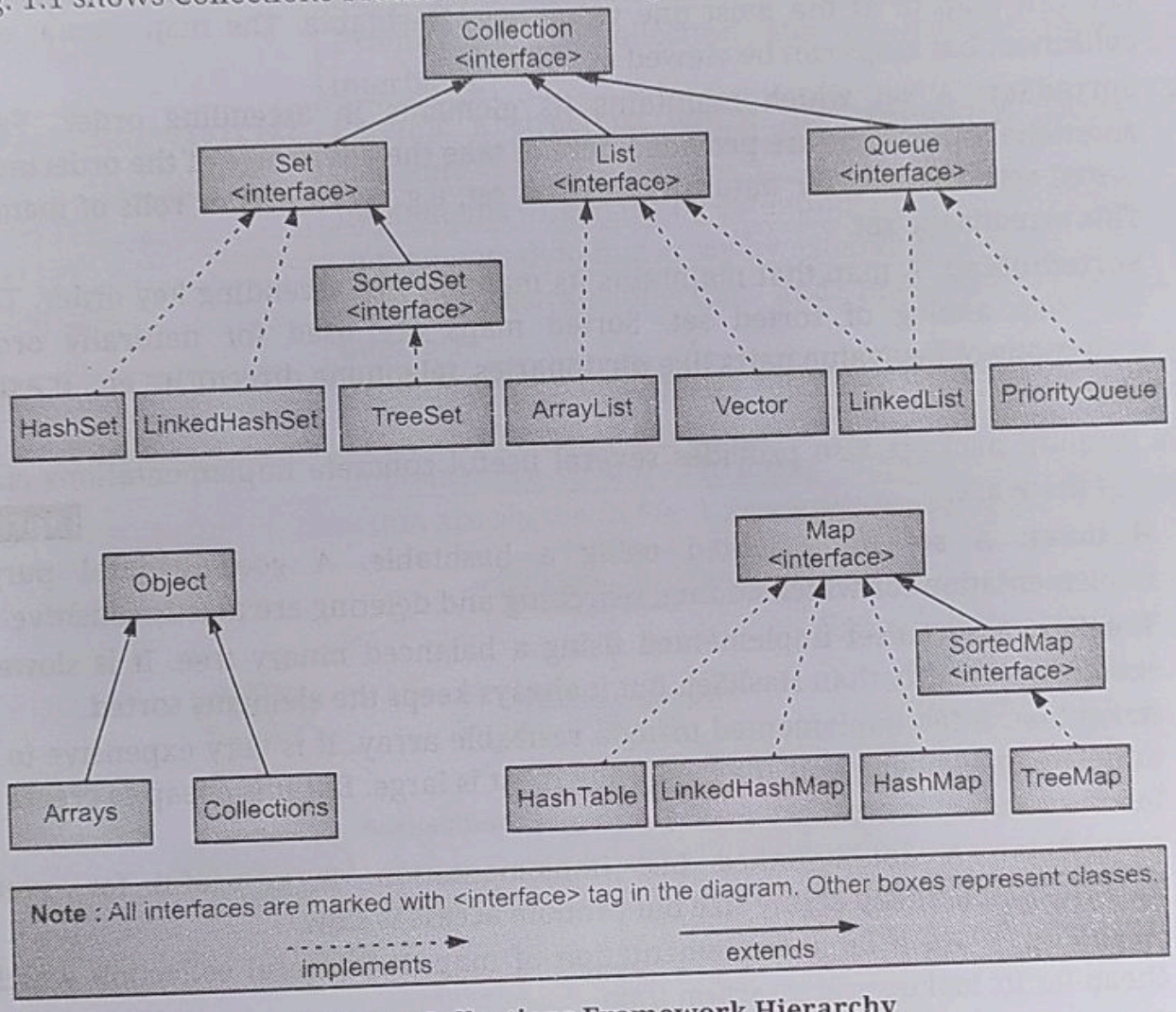

Fig. 1.1: Collections Framework Hierarchy

- The Collections Framework defines several interfaces. The collection interfaces are necessary because they determine the fundamental nature of the collection classes.

[April 18]

- The core collection interfaces are:
  1. **Collection:** It is a root of the collection hierarchy. A collection represents a group of objects, known as its elements. It enables us to work with groups of objects.
  2. **Set:** It is a collection that cannot contain the duplicate elements. It means that set handles unique elements. It extends collection.

3. **List:** List is an ordered collection called as sequence. List may contain duplicate elements. The user of a list generally has precise control over where the list each element is inserted and their integer index (positions), e.g. vector. This extends from collection.
4. **Queue:** The java.util.Queue interface is subtype of the java.util.Collection interface. It represents an ordered list of objects. A queue is designed to have elements inserted at the end of the queue and elements removed from the beginning of the queue. Just like a queue in a supermarket.
5. **Map:** An object that maps keys to values maps cannot contain duplicate keys. Each key can map to at the most one value. e.g. Hashtable. The map cannot extend collection, but maps can be viewed as collections.
6. **SortedSet:** A set which maintains its elements in ascending order. Various additional operations are provided here to take the advantage of the ordering. The sorted sets are used for naturally ordered set, e.g. word lists or rolls of members. This extends the set.
7. **SortedmMap:** A map that maintains its mappings in ascending key order. This is the map analog of sorted set. Sorted maps are used for naturally ordered collections of key/value pairs like dictionaries, telephone directories etc. It extends Map.

- The java.util package also provides several useful concrete implementations classes some of them are: **[April 18]**

1. **HashSet:** A set implemented using a hashtable. A good general purpose implementation for which adding, searching and deleting are mostly sensitive.
2. **TreeSet:** A sortedset implemented using a balanced binary tree. It is slower to search and modify than HashSet. But it always keeps the elements sorted.
3. **ArrayList:** A list implemented using a resizable array. It is very expensive to add or delete an element near the beginning if list is large. But it is cheap to create and fast for random access.
4. **LinkedList:** A doubly-linked List implementation. It is useful for queues. Modification is cheap at any size but random access is slow.
5. **HashMap:** A Hash table implementation of map. It is useful collection which is cheap for its lookup and insertion time.
6. **TreeMap:** It is an implementation of Sorted Map. It is using a balanced binary tree that keeps its elements ordered by key. It is very useful for ordered data set. Searching is done through key.

1.1.3 Collection Classes

- Collection classes are the utility classes. It has all static methods which are used to specify different applications. The methods takes parameter of no_threadsafe method and returns a thread safe method.

- The collection class defines a range of static utility method that operates on the collections. The utility methods are classified into two groups:
  1. Those methods that provide wrapped collections.
  2. Those methods that does not provide wrapped collections.
- The wrapped collections allow us to present different view of collections by synchronizing access to the collection or just by removing all the methods which could modify the collection.
- There are various collection classes which are shown below:

Class	Description
`AbstractCollection`	Implements various collection interfaces.
`AbstractList`	Extends AbstractCollection and implements various List interfaces.
`AbstractSequentialList`	Extends AbstractList which uses sequential access of its elements.
`LinkedList`	Implements linked list by extending AbstractSequentialList.
`ArrayList`	Implements dynamic array by extending AbstractList.
`AbstractSet`	Extends AbstractCollection and implements various set interface.
`HashSet`	Extends AbstractSet for use with hash table.
`LinkedHashSet`	Extends HashSet to allow insertion order iterations.
`TreeSet`	Implements a set stored in a tree. It extends the AbstractSet.

- The above classes of collection are shown in Fig. 1.2 as a full type tree for collections.

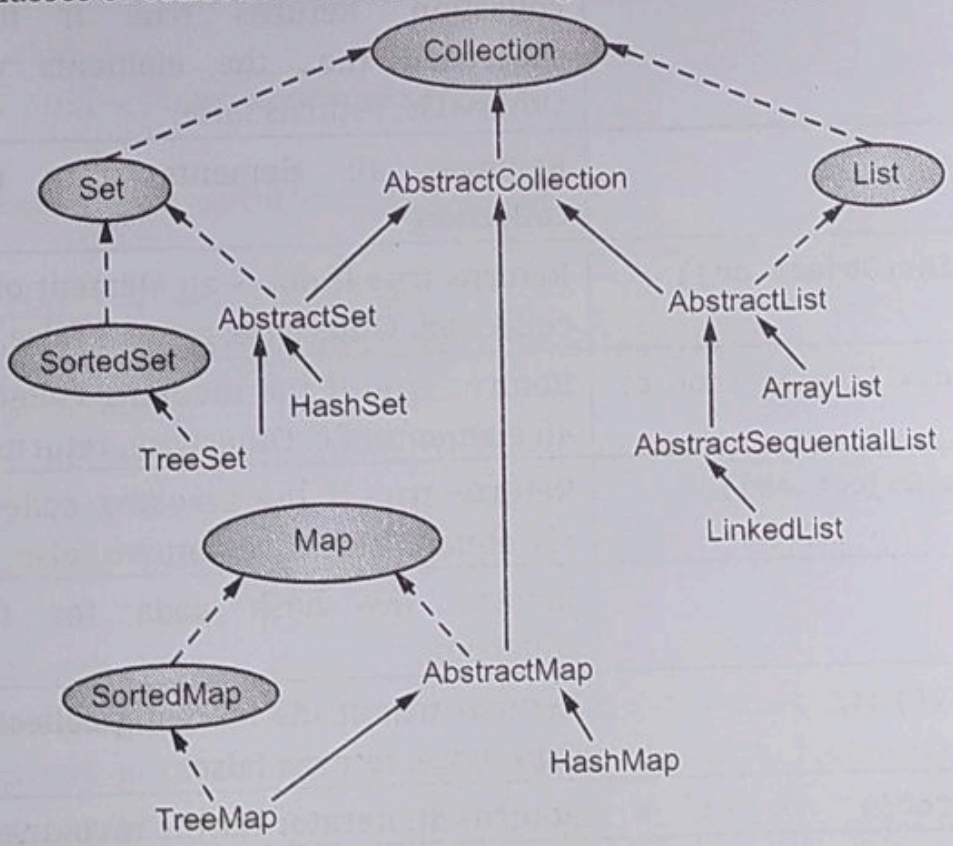

Fig. 1.2: Tree of Collection Classes

1.1.4 Collection Interface

- The Collection interface is used to pass around collections of objects where maximum generality is desired.
- For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument. This constructor, known as a conversion constructor, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's sub-interface or implementation type. In other words, it allows us to convert the collection's type.
- Suppose, for example, that we have a Collection<String> c, which may be a List, a Set or another kind of Collection. This idiom creates a new ArrayList (an implementation of the List interface), initially containing all the elements in c.

```
List<String> list = new ArrayList<String>(c);
```

Methods for Collection Interface:

Method	Description
`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.
`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 iterator()`	Returns an iterator for the invoking collection.

contd.

`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.
`Object[] toArray()`	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
`Object[] toArray(Object array[])`	Returns an array containing only those collection elements whose type matches that of array.

Program 1.1: Program for collection interface.

```
import java.util.*;
public class CollectionInterfaceExample
{
   public static void main(String[] args)
   {
      List list_1 = new ArrayList();
      List<String> list_2 = new ArrayList<String>();
      list_1.add(10);
      list_1.add(20);
      list_2.add("BTech");
      list_2.add("Smart");
      list_2.add("Class");
      list_1.addAll(list_2);
      System.out.println("Elements of list_1: " + list_1);
      System.out.println("Search for BTech: " + list_1.contains("BTech"));
      System.out.println("Search for list_2 in list_1: " +
                                          list_1.containsAll(list_2));
```

```
        System.out.println("Check whether list_1 and list_2 are equal: " +
                                                    list_1.equals(list_2));

        System.out.println("Check is list_1 empty: " + list_1.isEmpty());
        System.out.println("Size of list_1: " + list_1.size());
        System.out.println("Hashcode of list_1: " + list_1.hashCode());
        list_1.remove(0);
        System.out.println(list_1);
        list_1.retainAll(list_2);
        System.out.println(list_1);
        list_1.removeAll(list_2);
        System.out.println(list_1);
        list_2.clear();
        System.out.println(list_2);
    }
}
```

Output:

```
Elements of list_1: [10, 20, BTech, Smart, Class]
Search for BTech: true
Search for list_2 in list_1: true
Check whether list_1 and list_2 are equal: false
Check is list_1 empty: false
Size of list_1: 5
Hashcode of list_1: -764556324
[20, BTech, Smart, Class]
[BTech, Smart, Class]
[]
[]
```

1.2 LIST

- A List is an ordered Collection (sometimes called a sequence). The List interface extends Collection and declares the behavior of a collection that stores a sequence of elements.
- Lists may contain duplicate elements. Elements can be inserted or accessed by their position in the list, using a zero-based index.

Methods of List Interface:

Method	Description
`void add(int index, Object obj)`	Inserts obj into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
`Object get(int index)`	Returns the object stored at the specified index within the invoking collection.
`boolean addAll(int index, Collection c)`	Inserts all elements of c into the invoking list at the index passed in index. Any pre-existing 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.
`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 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.
`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. The 1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Program 1.2: Program for List interface.

```
import java.util.*;
public class ListDemo
{
    public static void main(String[] args)
    {
        List a1 = new ArrayList();
        a1.add("Zuber");
        a1.add("Mahesh");
        a1.add("Ayush");
        System.out.println(" ArrayList Elements:");
        System.out.print("\t" + a1);
        List l1 = new LinkedList();
        l1.add("Zuber ");
        l1.add("Mahesh ");
        l1.add("Ayush");
        System.out.println();
        System.out.println(" LinkedList Elements:");
        System.out.print("\t" + l1);
    }
}
```

Output:

```
ArrayList Elements:
    [Zuber, Mahesh, Ayush]
LinkedList Elements:
    [Zuber, Mahesh, Ayush]
```

1.2.1 ArrayList

- ArrayList class uses a dynamic array for storing the elements.
- It extends AbstractList class and implements List interface.

Constructors:

1. `ArrayList()` constructor builds an empty array list.
2. `ArrayList(Collection c)` constructor builds an array list that is initialized with the elements of the collection c.
3. `ArrayList(int capacity)` constructor builds an array list that has the specified initial capacity. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

Methods of ArrayList:

Method	Description
`void add(int index, Object element)`	Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index > size()).
`boolean add(Object o)`	Appends the specified element to the end of this list.
`boolean addAll(Collection c)`	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException if the specified collection is null.
`boolean addAll(int index, Collection c)`	Inserts all of the elements in the specified collection into this list, starting at the specified position. It throws NullPointerException if the specified collection is null.
`void clear()`	Removes all of the elements from this list.
`Object clone()`	Returns a shallow copy of this ArrayList.
`boolean contains(Object o)`	Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null? e==null: o.equals(e)).
`void ensureCapacity(int minCapacity)`	Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
`Object get(int index)`	Returns the element at the specified position in this list. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()).
`int indexOf(Object o)`	Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.

contd. ...

`int lastIndexOf(Object o)`	Returns the index in this list of the occurrence of the specified element, or -1 if list does not contain this element.
`Object remove(int index)`	Removes the element at the specified position this list. Throws IndexOutOfBoundsExceptio index out of range (index < 0 || index >= size()
`protected void removeRange(int fromIndex, int toIndex)`	Removes from this List all of the elements who index is between fromIndex, inclusive and toIndex, exclusive.
`Object set(int index, Object element)`	Replaces the element at the specified position this list with the specified element. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()).
`int size()`	Returns the number of elements in this list.
`Object[] toArray()`	Returns an array containing all of the elements this list in the correct order. Throws NullPointerException if the specified array is null.
`Object[] toArray(Object[] a)`	Returns an array containing all of the elements this list in the correct order; the runtime type of the returned array is that of the specified array
`void trimToSize()`	Trims the capacity of this ArrayList instance to the list's current size.

Program 1.3: Program for ArrayList.

```
import java.util.*;
class DemoArrayList
{
    public static void main(String args[])
    {
        ArrayList b = new ArrayList();
        b.add("Pune");
        b.add("Patna");
        b.add("Parbhani");
        b.add("Satara");
        System.out.println(b);
```

```
        b.remove("patna");
        System.out.println(b);
    }
}
```

Output:

```
[Pune, Patna, Parbhani, Satara]
[Pune, Patna, Parbhani, Satara]
```

1.2.2 LinkedList [April 18]

The LinkedList class extends AbstractSequentialList and implements the List interface. It provides a linked-list data structure.

Constructors:

1. `LinkedList()` constructor builds an empty linked list.
2. `LinkedList(Collection c)` constructor builds a linked list that is initialized with the elements of the collection c.

Methods for Linked List:

Method	Description
`void add(int index, Object element)`	Inserts the specified element at the specified position index in this list. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index > size()).
`boolean add(Object o)`	Appends the specified element to the end of this list.
`boolean addAll(Collection c)`	Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws NullPointerException if the specified collection is null.
`boolean addAll(int index, Collection c)`	Inserts all of the elements in the specified collection into this list, starting at the specified position. Throws NullPointerException if the specified collection is null.
`void addFirst(Object o)`	Inserts the given element at the beginning of this list.

contd. ...

`void addLast(Object o)`	Appends the given element to the end of this li
`void clear()`	Removes all of the elements from this list.
`Object clone()`	Returns a shallow copy of this LinkedList.
`boolean contains(Object o)`	Returns true if this list contains the specifi element. More formally, returns true if and o if this list contains at least one element e su that (o==null? e==null: o.equals(e)).
`Object get(int index)`	Returns the element at the specified position this list. Throws IndexOutOfBoundsException the specified index is is out of range (index < 0 index >= size()).
`Object getFirst()`	Returns the first element in this list. Throw NoSuchElementException if this list is empty.
`Object getLast()`	Returns the last element in this list. Throw NoSuchElementException if this list is empty.
`int indexOf(Object o)`	Returns the index in this list of the fir occurrence of the specified element, or -1 if t List does not contain this element.
`int lastIndexOf(Object o)`	Returns the index in this list of the la occurrence of the specified element, or -1 if th list does not contain this element.
`ListIterator listIterator(int index)`	Returns a list-iterator of the elements in this li (in proper sequence), starting at the specifie position in the list. Throw IndexOutOfBoundsException if the specifie index is is out of range (index < 0 || index > size()).
`Object remove(int index)`	Removes the element at the specified position i this list. Throws NoSuchElementException if thi list is empty.
`boolean remove(Object o)`	Removes the first occurrence of the specifie element in this list. Throw NoSuchElementException if this list is empty Throws IndexOutOfBoundsException if th specified index is is out of rang (index < 0 || index >= size()).

contd.

`Object removeFirst()`	Removes and returns the first element from this list. Throws NoSuchElementException if this list is empty.
`Object removeLast()`	Removes and returns the last element from this list. Throws NoSuchElementException if this list is empty.
`Object set(int index, Object element)`	Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0 || index >= size()).
`int size()`	Returns the number of elements in this list.
`Object[] toArray()`	Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.
`Object[] toArray(Object[] a)`	Returns an array containing all of the elements in this list in the correct order; the runtime type of the returned array is that of the specified array.

Program 1.4: Program for LinkedList.

```
import java.util.*;
class DemoLinkedlist
{
    public static void main(String args[])
    {
        LinkedList linkedlist1 = new LinkedList();
        linkedlist1.add("Item 2");
        linkedlist1.add("Item 3");
        linkedlist1.add("Item 4");
        linkedlist1.add("Item 5");
        linkedlist1.addFirst("Item 0");
        linkedlist1.addLast("Item 6");
        linkedlist1.add(1, "Item 1");
        System.out.println(linkedlist1);
        linkedlist1.remove("Item 6");
```

```
        System.out.println(linkedlist1);
        linkedlist1.removeLast();
        System.out.println(linkedlist1);
        System.out.println("\nUpdating linked list items");
        linkedlist1.set(0, "Red");
        linkedlist1.set(1, "Blue");
        linkedlist1.set(2, "Green");
        linkedlist1.set(3, "Yellow");
        linkedlist1.set(4, "Purple");
        System.out.println(linkedlist1);
    }
}
```

Output:

```
[Item 0, Item 1, Item 2, Item 3, Item 4, Item 5, Item 6]
[Item 0, Item 1, Item 2, Item 3, Item 4, Item 5]
[Item 0, Item 1, Item 2, Item 3, Item 4]
Updating linked list items
[Red, Blue, Green, Yellow, Purple]
```

Difference between ArrayList and LinkedList: [April 1

Sr. No.	ArrayList	LinkedList
1.	ArrayList internally uses dynamic array to store the elements.	LinkedList internally uses doubly linked list to store the elements.
2.	Manipulation with ArrayList is slow because it internally uses array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
3.	ArrayList class can act as a list only because it implements List only.	LinkedList class can act as a list and queue both because it implements List and Deque interfaces.
4.	ArrayList is better for storing and accessing data.	LinkedList is better for manipulating data.

1.2.3 Vector

[April 15

- The Vector class is analogous to ArrayList. It is a legacy class which is made to implement List and so it works as a collection.

- Vector implements a dynamic array. It is similar to ArrayList, but with two differences:
  1. Vector is synchronized.
  2. Vector contains many legacy methods that are not part of the collections framework.
- Vector proves to be very useful if we don't know the size of the array in advance or we just need one that can change sizes over the lifetime of a program.

Constructors:

1. `Vector()` constructor creates empty vector. Its size is 10 and capacity is 0. It is analogous to ArrayList.
2. `Vector(Collection c)` constructor creates a vector containing the elements of the collection c. The elements are accessed in the order returned by the collections iterator.
3. `Vector (int capacity)` constructor creates an empty vector with initial capacity.
4. `Vector (int capacity, int incr)` constructor creates an empty vector with initial capacity specified by capacity and capacity increment specified by incr.

Methods for Vector:

Method	Description
`void addElement(Object element)`	Inserts the element at the end of the Vector.
`int capacity()`	Returns the current capacity of the vector.
`int size()`	Returns the current size of the vector.
`void setSize(int size)`	Changes the existing size with the specified size.
`Boolean contains(Object element)`	Checks whether the specified element is present in the Vector. If the element is been found it returns true else false.
`Boolean containsAll(Collection c)`	Returns true if all the elements of collection c are present in the Vector.
`Object elementAt(int index)`	Returns the element present at the specified location in Vector.
`Object firstElement()`	Used for getting the first element of the vector.
`Object lastElement()`	Returns the last element of the array.
`Object get(int index)`	Returns the element at the specified index.
`boolean isEmpty()`	Returns true if Vector doesn't have any element.
`boolean removeElement(Object element)`	Removes the specified element from vector.
`boolean removeAll(Collection c)`	Removes all those elements from vector which are present in the Collection c.
`void setElementAt(Object element, int index)`	Updates the element of specifed index with the given element.

Program 1.5: Program for Vector.

```
import java.util.*;
class VectorDemo
{
      public static void main(String args[])
      {
         Vector vector = new Vector(5);
         System.out.println("Size: " + vector.size());
         System.out.println("Capacity: " + vector.capacity());
         vector.addElement(new Integer(0));
         vector.addElement(new Integer(1));
         vector.addElement(new Integer(2));
         vector.addElement(new Integer(3));
         vector.addElement(new Integer(4));
         vector.addElement(new Integer(5));
         vector.addElement(new Integer(6));
         vector.addElement(new Integer(5));
         vector.addElement(new Integer(6));
         vector.addElement(new Double(3.14159));
         vector.addElement(new Float(3.14159));
         System.out.println("Capacity: " + vector.capacity());
         System.out.println("Size: " + vector.size());
         System.out.println(vector);
         System.out.println("First item:"+Integer)vector.firstElement());
         System.out.println("Last item: "+(Float) vector.lastElement());
         if(vector.contains(new Integer(3)))
         System.out.println("Found 3");
      }
}
```

Output:

```
Size: 0
Capacity: 5
Capacity: 20
Size: 11
[0, 1, 2, 3, 4, 5, 6, 5, 6, 3.14159, 3.14159]
First item:0
Last item: 3.14159
Found 3
```

1.2.4 Stack

- Stack is a subclass of Vector that implements a standard Last-In, First-Out (LIFO) stack.
- Stack only defines the default constructor Stack(), which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

Methods for Stack:

Method	Description
`boolean empty()`	Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
`Object peek()`	Returns the element on the top of the stack, but does not remove it.
`Object pop()`	Returns the element on the top of the stack, removing it in the process.
`Object push(Object element)`	Pushes element onto the stack. element is also returned.
`int search(Object element)`	Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned.

Program 1.6: Program for Stack.

```
import java.util.*;
class DemoStack
{
   public static void main(String args[])
   {
      Stack stack1 = new Stack();
      try
      {
         stack1.push(new Integer(0));
         stack1.push(new Integer(1));
         stack1.push(new Integer(2));
         stack1.push(new Integer(3));
         stack1.push(new Integer(4));
         stack1.push(new Integer(5));
         stack1.push(new Integer(6));
         System.out.println("Pop->"+(Integer) stack1.pop());
         System.out.println("Pop->"+(Integer) stack1.pop());
```

```
            System.out.println("Pop->"+(Integer) stack1.pop());
            System.out.println("Pop->"+(Integer) stack1.pop());
            System.out.println("Pop->"+(Integer) stack1.pop());
            System.out.println("Pop->"+(Integer) stack1.pop());
            System.out.println("Pop->"+(Integer) stack1.pop());
        }
            catch (EmptyStackException e) {}
        }
    }
```

Output:

```
Pop->6
Pop->5
Pop->4
Pop->3
Pop->2
Pop->1
Pop->0
```

1.2.5 Queue

- Queue interface is in java.util package of java.Queue is a linear collection that supports element insertion and removal at both ends.
- A Queue interface extends the Collection interface to define an ordered collection for holding elements in a FIFO (first-in-first-out) manner to process them i.e. in a FIFO queue the element that is inserted firstly will also get removed first. Queue does not require any fix dimension like String array and int array.
- Besides basic collection operations, queues also provide additional operations such as insertion, removal, and inspection.

Methods for Queue:

Method	Description
`poll()`	Returns and removes the element at the front end of the container.
`remove()`	Removes element at the head of the queue and throws NoSuchElementException if queue is empty.
`peek()`	Returns the element at the head of the queue without removing it. Returns null if queue is empty.
`element()`	Same as peek(), but throws NoSuchElementException if queue is empty.
`offer(E obj)`	Adds object to queue. This method is preferable to add() method since this method does not throws an exception when the capacity of the container is full since it returns false.

- The remove() and poll() methods differ only in their behavior when the queue is empty: the remove() method throws an exception, while the poll() method returns null.

Program 1.7: In following program we use the LinkedList class which implements the Queue interface and add items to it using both the add() and offer() methods.

```
import java.util.*;
public class DemoQueue
{
   public static void main(String[] args)
   {
      Queue oqueue=new LinkedList();
      oqueue.add("1");
      oqueue.add("2");
      oqueue.add("3");
      oqueue.add("4");
      oqueue.add("5");
      Iterator itr=oqueue.iterator();
      System.out.println("Initial Size of Queue:"+oqueue.size());
      while(itr.hasNext())
      {
         String iteratorValue=(String)itr.next();
         System.out.println("Queue Next Value:"+iteratorValue);
      }
      // get value and does not remove element from queue
      System.out.println("Queue peek:"+oqueue.peek());
      // get first value and remove that object from queue
      System.out.println("Queue poll:"+oqueue.poll());
      System.out.println("Final Size of Queue:"+oqueue.size());
   }
}
```

Output:

```
Initial Size of Queue:5
Queue Next Value:1
Queue Next Value:2
Queue Next Value:3
Queue Next Value:4
Queue Next Value:5
Queue peek:1
Queue poll:1
Final Size of Queue:4
```

1.3 SET

- Set interface defines a Set. 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.
- Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

Methods for Set:

Method	Description
`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.
`isEmpty()`	Returns true if the collection has no elements.
`Iterator iterator()`	Returns an Iterator object for the collection which may be used to retrieve an object.
`remove()`	Removes a specified object from the collection.
`size()`	Returns the number of elements in the collection.

Program 1.8: Program for Set.

```
import java.util.*;
public class DemoSet
{
   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){}
    }
}
```

Output:

```
[34, 22, 10, 60, 30]
The sorted list is:
[10, 22, 30, 34, 60]
The First element of the set is: 10
The last element of the set is: 60
```

1.3.1 HashSet [April 16]

- HashSet class extends AbstractSet and implements the Set interface.
- It creates a collection that uses a hash table for storage.
- Hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code.
- The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

Constructors: [April 16]

1. `HashSet()` constructor form constructs a default hash set.
2. `HashSet(Collection c)` constructor initializes the hash set by using the elements of c.
3. `HashSet(int capacity)`constructor initializes the capacity of the hash set to capacity. The capacity grows automatically as elements are added to the Hash.
4. `HashSet(int capacity, float fillRatio)` initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments.

Methods for HashSet:

Method	Description
`boolean add(Object o)`	Adds the specified element to this set if it is not alre present.
`void clear()`	Removes all of the elements from this set.
`Object clone()`	Returns a shallow copy of this HashSet instance: elements themselves are not cloned.
`boolean contains(Object o)`	Returns true if this set contains the specified element.
`boolean isEmpty()`	Returns true if this set contains no elements.
`Iterator iterator()`	Returns an iterator over the elements in this set.
`boolean remove(Object o)`	Removes the specified element from this set if it is prese
`int size()`	Returns the number of elements in this set (its cardinalit

Program 1.9: Program for HashSet (output in sorted order).

```
import java.util.*;
class DemoHashset
{
    public static void main(String args[])
    {
        hashSet hashset1 = new HashSet();
        hashset1.add("Item 0");
        hashset1.add("Item 1");
        hashset1.add("Item 2");
        hashset1.add("Item 3");
        hashset1.add("Item 4");
        hashset1.add("Item 5");
        hashset1.add("Item 5");
        hashset1.add("Item 6");
        hashset1.add("Item 6");
        hashset1.add("Item 6");
        System.out.println(hashset1);
    }
}
```

Output:

```
[Item 0, Item 4, Item 3, Item 2, Item 1, Item 6, Item 5]
```

3.2 TreeSet

[April 17, 18; Oct. 17]

TreeSet class provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.

Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

onstructors:

1. `TreeSet()` constructor constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements.
2. `TreeSet(Collection c)` constructor builds a tree set that contains the elements of c.
3. `TreeSet(Comparator comp)` constructs an empty tree set that will be sorted according to the comparator specified by comp.
4. `TreeSet(SortedSet ss)` builds a tree set that contains the elements of ss.

Methods for TreeSet:

Method	Description
`void add(Object o)`	Adds the specified element to this set if it is not already present.
`boolean addAll(Collection c)`	Adds all of the elements in the specified collection to this set.
`void clear()`	Removes all of the elements from this set.
`Object clone()`	Returns a shallow copy of this TreeSet instance.
`Comparator comparator()`	Returns the comparator used to order this sorted set, or null if this tree set uses its elements natural ordering.
`boolean contains(Object o)`	Returns true if this set contains the specified element.
`Object first()`	Returns the first (lowest) element currently in this sorted set.
`SortedSet headSet(Object toElement)`	Returns a view of the portion of this set whose elements are strictly less than toElement.
`boolean isEmpty()`	Returns true if this set contains no elements.
`Iterator iterator()`	Returns an iterator over the elements in this set.
`Object last()`	Returns the last (highest) element currently in this sorted set.
`boolean remove(Object o)`	Removes the specified element from this set if it is present.
`int size()`	Returns the number of elements in this set (its cardinality).
`SortedSet subSet(Object fromElement, Object toElement)`	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
`SortedSet tailSet(Object fromElement)`	Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

Program 1.10: Program for TreeSet.

```
import java.util.*;
class DemoTreeSet
{
   public static void main(String args[])
   {
      TreeSet treeset1 = new TreeSet();
      treeset1.add("Monday");
      treeset1.add("Tuesday");
      treeset1.add("Wednesday");
      treeset1.add("Thursday");
      treeset1.add("Friday");
      treeset1.add("Saturday");
      treeset1.add("Sunday");
      System.out.println(treeset1);
   }
}
```

Output:

```
[Friday, Monday, Saturday, Sunday, Thursday, Tuesday, Wednesday]
```

1.3.3 LinkedHashSet

- This class extends HashSet, but adds no members of its own. LinkedHashSet maintai a linked list of the entries in the set, in the order in which they were inserted. T allows insertion-order iteration over the set.
- LinkedHashSet is also an implementation of Set interface, it is similar to the Hash and TreeSet except the below mentioned differences:
  1. `HashSet` doesn't maintain any kind of order of its elements.
  2. `TreeSet` sorts the elements in ascending order.
  3. `LinkedHashSet` maintains the insertion order. Elements get sorted in the sa sequence in which they have been added to the Set.

Constructors:

1. `LinkedHashSet()` constructs a default hash set.
2. `LinkedHashSet(Collection c)` initializes the hash set by using the elements of
3. `LinkedHashSet(int capacity)` initializes the capacity of the hash set to capa The capacity grows automatically as elements are added to the Hash.
4. `LinkedHashSet(int capacity, float fillRatio)` initializes both the capacity the fill ratio (also called load capacity) of the hash set from its arguments

rogram 1.11: Program for LinkedHashSet.

```
import java.util.*;
public class DemoLinkedHashSet
{
    public static void main(String args[])
    {
        LinkedHashSet hs = new LinkedHashSet();        // create a hash set
        // add elements to the hash set
        hs.add("B");
        hs.add("A");
        hs.add("D");
        hs.add("E");
        hs.add("C");
        hs.add("F");
        System.out.println(hs);
    }
}
```

utput:

```
[B, A, D, E, C, F]
```

1.4 MAP INTERFACE [April 17]

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.

There are following three main implementations of Map interfaces:

1. **HashMap:** It makes no guarantees concerning the order of iteration
2. **TreeMap:** It stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashMap.
3. **LinkedHashMap:** It orders its elements based on the order in which they were inserted into the set (insertion-order).

Methods for Map:

Method	Description
`void clear()`	Removes all key/value pairs from the invoking m
`boolean containsKey(Object k)`	Returns true if the invoking map contains k as a k Otherwise, returns false.
`boolean containsValue(Object v)`	Returns true if the map contains v as a val Otherwise, returns false.
`Set entrySet()`	Returns a Set that contains the entries in the m The set contains objects of type Map.Entry. T method provides a set-view of the invoking map.
`boolean equals(Object obj)`	Returns true if obj is a Map and contains the sa entries. Otherwise, returns false.
`Object get(Object k)`	Returns the value associated with the key k.
`int hashCode()`	Returns the hash code for the invoking map.
`boolean isEmpty()`	Returns true if the invoking map is emp Otherwise, returns false.
`Set keySet()`	Returns a Set that contains the keys in the invoki map. This method provides a set-view of the keys the invoking map.
`Object put(Object k, Object v)`	Puts an entry in the invoking map, overwriting a previous value associated with the key. The key a value are k and v, respectively. Returns null if t key did not already exist. Otherwise, the previo 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 t map. This method provides a collection-view of t values in the map.

Program 1.12: Program for Map interface.

```
import java.util.*;
public class DemoMap
{
    public static void main(String[] args)
```

```
    {
        Map m1 = new HashMap();
        m1.put("Rudra", "8");
        m1.put("Mahi", "31");
        m1.put("Ayush", "12");
        m1.put("Dhruthik", "14");
        m1.put("Atharva", "14");
        System.out.println();
        System.out.println(" Map Elements");
        System.out.print("\t" + m1);
    }
}
```

Output:

```
Map Elements
    {Mahi=31, Dhruthik=14, Rudra=8, Ayush=12, Atharva=14}
```

1.4.1 HashMap

- HashMap contains values based on the key. This class implements the Map interface and extends AbstractMap class.
- It contains only unique elements. It may have one null key and multiple null values. It maintains no order.

Constructors:

1. `HashMap()` constructs a default hash map.
2. `HashMap(Map m)` initializes the hash map by using the elements of m.
3. `HashMap(int capacity)` initializes the capacity of the hash map to capacity.
4. `HashMap(int capacity, float fillRatio)` initializes both the capacity and fill ratio of the hash map by using its arguments.

Methods for HashMap:

Method	Description
`void clear()`	Removes all mappings from the map.
`Object clone()`	Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
`boolean containsKey(Object key)`	Returns true if this map contains a mapping for the specified key.
`boolean containsValue(Object value)`	Returns true if this map maps one or more keys to the specified value.

contd. ...

`Set entrySet()`	Returns a collection view of the mappings contai in this map.
`Object get(Object key)`	Returns the value to which the specified key mapped in this identity hash map, or null if the m contains no mapping for this key.
`boolean isEmpty()`	Returns true if this map contains no key-val mappings.
`Set keySet()`	Returns a set view of the keys contained in this ma
`Object put(Object key, Object value)`	Associates the specified value with the specified k in this map.
`putAll(Map m)`	Copies all of the mappings from the specified map this map These mappings will replace any mappin that this map had for any of the keys currently the specified map.
`Object remove(Object key)`	Removes the mapping for this key from this map present.
`int size()`	Returns the number of key-value mappings in th map.
`Collection values()`	Returns a collection view of the values contained this map.

Program 1.13: Program for HashMap.

```
import java.util.*;
import java.io.*;
public class DemoHashMap
{
   public static void main(String args[])throws IOException
   {
      HashMap hm = new HashMap();        // Create a HashMap
      // Put elements to the map
      hm.put("Ziva", new Double(3434.34));
      hm.put("Malhar", new Double(123.22));
      hm.put("Adhira", new Double(1378.00));
      hm.put("Kismat", new Double(99.22));
      hm.put("Akash", new Double(-19.08));
      // Get a set of the entries
```

```
        Set set = hm.entrySet();
        // Get an iterator
        Iterator i = set.iterator();
        // Display elements
        while(i.hasNext())
        {
            Map.Entry me = (Map.Entry)i.next();
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();
        // Deposit 1000 into Zara's account
        double balance = ((Double)hm.get("Ziva")).doubleValue();
        hm.put("Ziva", new Double(balance + 1000));
        System.out.println("Ziva's new balance: " + hm.get("Ziva"));
    }
}
```

Output:

```
Ziva: 3434.34
Kismat: 99.22
Akash: -19.08
Adhira: 1378.0
Malhar: 123.22
Ziva's new balance: 4434.34
```

1.4.2 LinkedHashMap

- LinkedHashMap class extends HashMap and maintains a linked list of the entries in the map, in the order in which they were inserted.
- This allows insertion-order iteration over the map. That is, when iterating a LinkedHashMap, the elements will be returned in the order in which they were inserted.
- We can also create a LinkedHashMap that returns its elements in the order in which they were last accessed.

Constructors:

1. `LinkedHashMap()` constructs a default LinkedHashMap.
2. `LinkedHashMap(Map m)` initializes the LinkedHashMap with the elements from m.
3. `LinkedHashMap(int capacity)` initializes the capacity.

4. `LinkedHashMap(int capacity, float fillRatio)` initializes both capacity and ratio. The meaning of capacity and fill ratio are the same as for HashMap.
5. `LinkedHashMap(int capacity, float fillRatio, boolean Order)` allows us specify whether the elements will be stored in the linked list by insertion order, by order of last access. If Order is true, then access order is used. If Order is fa then insertion order is used.

Methods:

1. `void clear()` method removes all mappings from this map.
2. `boolean containsKey(Object key)` method returns true if this map maps one more keys to the specified value.
3. `Object get(Object key)` method returns the value to which this map maps specified key.
4. `protected boolean removeEldestEntry(Map.Entry eldest)` method returns tr if this map should remove its eldest entry.

Program 1.14: Program for LinkedHashMap.

```
import java.util.*;
class DemoLinkedHashMap
{
   public static void main(String args[])
   {
      LinkedHashMap<Integer, String> lhmap = new LinkedHashMap<Integer,
                                                              String>(
      lhmap.put(500,"Goa");
      lhmap.put(510,"Pune");
      lhmap.put(520,"Delhi");
      System.out.println("Keys: "+lhmap.keySet());      //Fetching key
      System.out.println("Values: "+lhmap.values());    //Fetching value
      System.out.println("Key-Value pairs: "+lhmap.entrySet()); //Fetchi
                                                              key-value pa
   }
}
```

Output:

```
Keys: [500, 510, 520]
Values: [Goa, Pune, Delhi]
Key-Value pairs: [500=Goa, 510=Pune, 520=Delhi]
```

1.4.3 Hashtable

[Oct. 17, April 19]

- Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.
- A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method.
- A Hashtable contains values based on the key. It implements the Map interface and extends Dictionary class.
- It contains only unique elements. It may have not have any null key or value. It is synchronized.

Constructors:

1. `Hashtable()` default constructor.
2. `Hashtable(int size)` creates a hash table that has an initial size specified by size.
3. `Hashtable(int size, float fillRatio)` creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward.
4. `Hashtable(Map m)` creates a hash table that is initialized with the elements in m. The capacity of the hash table is set to twice the number of elements in m. The default load factor of 0.75 is used.

Methods for HashTable:

Method	Description
`void clear()`	Resets and empties the hash table.
`Object clone()`	Returns a duplicate of the invoking object.
`boolean contains(Object value)`	Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found.
`boolean containsKey(Object key)`	Returns true if some key equal to key exists within the hash table. Returns false if the key isn't found.
`boolean containsValue(Object value)`	Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found.
`Enumeration elements()`	Returns an enumeration of the values contained in the hash table.
`Object get(Object key)`	Returns the object that contains the value associated with key. If key is not in the hash table, a null object is returned.

contd. ...

`boolean isEmpty()`	Returns true if the hash table is empty; returns false if it contains at least one key.
`Enumeration keys()`	Returns an enumeration of the keys contained in the hash table.
`Object put(Object key, Object value)`	Inserts a key and a value into the hash table. Returns null if key isn't already in the hash table; returns the previous value associated with key if key is already in the hash table.
`void rehash()`	Increases the size of the hash table and rehashes all of its keys.
`Object remove(Object key)`	Removes key and its value. Returns the value associated with key. If key is not in the hash table, a null object is returned.
`int size()`	Returns the number of entries in the hash table.
`String toString()`	Returns the string equivalent of a hash table.

Program 1.15: Program for Hashtable.

```
import java.util.*;
class DemoHashtable
{
   public static void main(String args[])
   {
      Hashtable<Integer,String> mt=new Hashtable<Integer,String>();
      mt.put(1,"January");
      mt.put(2,"February");
      mt.put(4,"April");
      mt.put(5,"March");
      System.out.println("Initial Hashtable: "+mt);
      mt.put(3,"March");  //Inserts, as the pair
      System.out.println("Updated Hashtable: "+mt);
   }
}
```

Output:

```
Initial Hashtable: {5=March, 4=April, 2=February, 1=January}
Updated Hashtable: {5=March, 4=April, 3=March, 2=February, 1=January}
```

1.4.4 TreeMap [Oct. 18]

- The TreeMap class implements the Map interface by using a tree. A TreeMap provides an efficient means of storing key/value pairs in sorted order, and allows rapid retrieval.
- TreeMap class implements Map interface similar to HashMap class. The main difference between them is that HashMap is an unordered collection while TreeMap is sorted in the ascending order of its keys.
- TreeMap is unsynchronized collection class which means it is not suitable for thread-safe operations until unless synchronized explicitly.

Constructors:

1. `TreeMap()` constructs an empty tree map that will be sorted by using the natural order of its keys.
2. `TreeMap(Comparator comp)` constructs an empty tree-based map that will be sorted by using the Comparator comp.
3. `TreeMap(Map m)` initializes a tree map with the entries from m, which will be sorted by using the natural order of the keys.
4. `TreeMap(SortedMap sm)` initializes a tree map with the entries from sm, which will be sorted in the same order as sm.

Methods for TreeMap:

Method	Description
`void clear()`	Removes all mappings from this TreeMap.
`Object clone()`	Returns a shallow copy of this TreeMap instance.
`Comparator comparator()`	Returns the comparator used to order this map, or null if this map uses its keys' natural order.
`boolean containsKey(Object key)`	Returns true if this map contains a mapping for the specified key.
`boolean containsValue(Object value)`	Returns true if this map maps one or more keys to the specified value.
`Set entrySet()`	Returns a set view of the mappings contained in this map.
`Object firstKey()`	Returns the first (lowest) key currently in this sorted map.
`Object get(Object key)`	Returns the value to which this map maps the specified key.

contd. ...

`SortedMap headMap(Object toKey)`	Returns a view of the portion of this map whose keys are strictly less than toKey.
`Set keySet()`	Returns a Set view of the keys contained in this map.
`Object lastKey()`	Returns the last (highest) key currently in this sorted map.
`Object put(Object key, Object value)`	Associates the specified value with the specified key in this map.
`void putAll(Map map)`	Copies all of the mappings from the specified map to this map.
`Object remove(Object key)`	Removes the mapping for this key from this TreeMap if present.
`int size()`	Returns the number of key-value mappings in this map.
`SortedMap subMap(Object fromKey, Object toKey)`	Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
`SortedMap tailMap(Object fromKey)`	Returns a view of the portion of this map whose keys are greater than or equal to fromKey.
`Collection values()`	Returns a collection view of the values contained in this map.

Program 1.16: Program for TreeMap.

```
import java.util.*;
public class DemoTreeMap
{
    public static void main(String args[])
    {
        TreeMap<Integer,String> map=new TreeMap<Integer,String>();
        map.put(1,"Tiger");
        map.put(2,"Lion");
        map.put(5,"Monkey");
        map.put(3,"Elephant");
        System.out.println("Initial TreeMap");
        for(Map.Entry m:map.entrySet())
        {
            System.out.println(m.getKey()+" "+m.getValue());
        }
```

```
        map.remove(5);
        System.out.println("After remove() method");
        for(Map.Entry m:map.entrySet())
        {
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

Output:

```
Initial TreeMap
1 Tiger
2 Lion
3 Elephant
5 Monkey
After remove() method
1 Tiger
2 Lion
3 Elephant
```

1.5 COMPARATOR INTERFACE

- Comparator interface is used to order the objects of user-defined class.
- Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what sorted order means.
- The Comparator interface defines two methods i.e., compare( ) and equals( ).

1. **The compare() Method:**

- The compare() method, shown here, compares two elements for order.

```
int compare(Object obj1, Object obj2)
```

where, obj1 and obj2 are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if obj1 is greater than obj2. Otherwise, a negative value is returned.

- By overriding compare(), we can alter the way that objects are ordered. For example, to sort in reverse order, we can create a comparator that reverses the outcome of a comparison.

2. **The equals() Method:**

- The equals() method, shown here, tests whether an object equals the invoking comparator:

```
boolean equals(Object obj)
```

where, obj is the object to be tested for equality. The method returns true if obj and invoking object are both Comparator objects and use the same ordering. Otherwise returns false.

- Overriding equals() is unnecessary, and most simple comparators will not do so.

Program 1.17: Program for comparator interface.

```
import java.util.*;
class Horse implements Comparator<Horse>, Comparable<Horse>
{
    private String name;
    private int age;
    Horse()
    {
    }
    Horse(String n, int a)
    {
        name = n;
        age = a;
    }
    public String getHorseName()
    {
        return name;
    }
    public int getHorseAge()
    {
        return age;
    }
    // Overriding the compareTo method
    public int compareTo(Horse h)
    {
        return (this.name).compareTo(h.name);
    }
    // Overriding the compare method to sort the age
    public int compare(Horse h, Horse h1)
    {
        return h.age - h1.age;
    }
}
```

```
public class DemoComparator
{
   public static void main(String args[])
   {
      // Takes a list o Horse objects
      List<Horse> list = new ArrayList<Horse>();
      list.add(new Horse("Shaggy",3));
      list.add(new Horse("Lacy",2));
      list.add(new Horse("Roger",10));
      list.add(new Horse("Tommy",4));
      list.add(new Horse("Tammy",1));
      Collections.sort(list);// Sorts the array list
      for(Horse a: list)//printing the sorted list of names
      System.out.print(a.getHorseName() + ", ");
      // Sorts the array list using comparator
      Collections.sort(list, new Horse());
      System.out.println(" ");
      for(Horse a: list)                    //printing the sorted list of ages
      System.out.print(a.getHorseName() +": "+
      a.getHorseAge() + ", ");
   }
}
```

Output:

```
Lacy, Roger, Shaggy, Tammy, Tommy,
Tammy: 1, Lacy: 2, Shaggy: 3, Tommy: 4, Roger: 10,
```

1.5.1 Iterator

[April 16, 17, 18]

- Iterator is used to iterate through elements of a collection class. Using Iterator we can traverse in one direction (forward).
- When, we will want to cycle through the elements in a collection. For example, we might want to display each element.
- The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.
- Iterator enables us to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

- Before we can access a collection through an iterator, you must obtain one. Each of th collection classes provides an iterator() method that returns an iterator to the start the collection.
- By using this iterator object, we can access each element in the collection, one eleme at a time.
- In general, to use an iterator to cycle through the contents of a collection, follo following steps:
  1. Obtain an iterator to the start of the collection by calling the collection's iterator method.
  2. Set up a loop that makes a call to hasNext(). Have the loop iterate as long hasNext( ) returns true.
  3. Within the loop, obtain each element by calling next().
- For collections that implement List, you can also obtain an iterator by calli ListIterator.

Methods:

1. `boolean hasNext()` method returns true if there are more elements. Otherwi returns false.
2. `Object next()` method returns the next element. Thro NoSuchElementException if there is not a next element.
3. `void remove()` method removes the current element. Thro IllegalStateException if an attempt is made to call remove( ) that is not preceded a call to next( ).

- Iterator is used for iterating (looping) various collection classes such as HashMa ArrayList, LinkedList etc.

Program 1.18: Program for iterator.

```
import java.util.ArrayList;
import java.util.Iterator;
public class DemoIterator
{
    public static void main(String args[])
    {
        ArrayList names = new ArrayList();
        names.add("Chaitanya");
        names.add("Shiva");
        names.add("Jai");
        Iterator it = names.iterator();
```

```
        while(it.hasNext())
        {
            String obj = (String)it.next();
            System.out.println(obj);
        }
    }
}
```

tput:

```
Chaitanya
Shiva
Jai
```

.2 ListIterator [April 16, 17]

ListIterator used to iterate through elements of a collection class.

Using ListIterator we can traverse the collection class on both the directions i.e., backward and forward.

hods for ListIterator:

Method	Description
id add(Object obj)	Inserts obj into the list in front of the element that will be returned by the next call to next().
olean hasNext()	Returns true if there is a next element. Otherwise, returns false.
olean hasPrevious()	Returns true if there is a previous element. Otherwise, returns false.
ject next()	Returns the next element. A NoSuchElementException is thrown if there is not a next element.
t nextIndex()	Returns the index of the next element. If there is not a next element, returns the size of the list.
ject previous()	Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.
t previousIndex()	Returns the index of the previous element. If there is not a previous element, returns -1.
id remove()	Removes the current element from the list. An IllegalStateException is thrown if remove() is called before next() or previous() is invoked.
id set(Object obj)	Assigns obj to the current element. This is the element last returned by a call to either next() or previous().

Program 1.19: Program of ListIterator.

```
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;
public class DemoListIterator
{
   public static void main(String a[])
   {
      ListIterator<String> litr = null;
      List<String> names = new ArrayList<String>();
      names.add("Shrimant");
      names.add("Ruha");
      names.add("Prapti");
      names.add("Tanaji");
      names.add("Kavya");
      //Obtaining list iterator
      litr=names.listIterator();
      System.out.println("Traversing the list in forward direction:")
      while(litr.hasNext())
      {
         System.out.println(litr.next());
      }
      System.out.println("\nTraversing the list in backward direction
      while(litr.hasPrevious())
      {
         System.out.println(litr.previous());
      }
   }
}
```

Output:

```
Traversing the list in forward direction:
Shrimant
Ruha
Prapti
Tanaji
Kavya
```

```
Traversing the list in backward direction:
Kavya
Tanaji
Prapti
Ruha
Shrimant
```

.5.3 Enumeration Interface

The Enumeration interface defines the methods by which you can enumerate, (obtain one at a time) the elements in a collection of objects.

This legacy interface has been superceded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code.

However, it is used by several methods defined by the legacy classes such as Vector and Properties, is used by several other API classes, and is currently in widespread use in application code.

ethods:

1. `boolean hasMoreElements()`: When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.
2. `Object nextElement()`: This returns the next object in the enumeration as a generic Object reference.

rogram 1.20: Program for Enumeration Interface.

```
import java.util.Vector;
import java.util.Enumeration;
public class DemoEnumeration
{
   public static void main(String args[])
   {
      Enumeration days;
      Vector dayNames = new Vector();
      dayNames.add("Sunday");
      dayNames.add("Monday");
      dayNames.add("Tuesday");
      dayNames.add("Wednesday");
      dayNames.add("Thursday");
      dayNames.add("Friday");
      dayNames.add("Saturday");
      days = dayNames.elements();
```

```
        while(days.hasMoreElements())
        {
            System.out.println(days.nextElement());
        }
    }
}
```

Output:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

ADDITIONAL PROGRAMS

Program 1: Program for ArrayList and methods interface.

```
import java.util.*;
class ArrayListDemo
{
    public static void main (String args[])
    {
        ArrayList a1 = new ArrayList();
        System.out.println("Initial size of ArrayList " + a1.size());
        a1.add("First");
        a1.add("Second");
        a1.add("Third");
        a1.add("Fourth");
        a1.add("Fifth");
        a1.add(2, "Middle");
        System.out.println("Size of ArrayList after addition is " +
                                                        a1.size
        System.out.println("Elements in ArrayList are " + a1);
        a1.remove("Third");
        a1.remove(1);
        System.out.println("Size of ArrayList after deletion is " +
                                                        a1.size
        System.out.println("Elements in ArrayList are " + a1);
    }
}
```

Output:

```
Initial size of ArrayList 0
Size of ArrayList after addition is 6
Elements in ArrayList are [First, Second, Middle, Third, Fourth, Fifth]
Size of ArrayList after deletion is 4
Elements in ArrayList are [First, Middle, Fourth, Fifth]
```

Program 2: Program to display the sum of ArrayList elements.

```
import java.util.*;
import java.io.*;
class ArrayListToArray
{
   public static void main (String args[])throws IOException
   {
      ArrayList <Integer>a = new ArrayList <Integer>();
      a.add(11);
      a.add(22);
      a.add(33);
      a.add(44);
      a.add(55);
      System.out.println("Contents of ArrayList are"+ a);
      int[] ia = new int[a.size()];
      int sum = 0;
      for(int i = 0; i < a.size(); i++)
      {
         ia[i] = a.get(i).intValue();
         sum+= ia[i];
      }
      System.out.println("Sum of the elements is " + sum);
   }
}
```

Output:

```
Contents of ArrayList are[11, 22, 33, 44, 55]
Sum of the elements is 165
```