CET1042B: Object Oriented Programming with C++ and Java

SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY

S. Y. B. TECH. COMPUTER SCIENCE AND ENGINEERING
(CYBERSECURITY AND FORENSICS)

CET1042B: Object Oriented Programming with C++ and Java

Teaching Scheme
Theory: 2 Hrs. / Week

Credits: 02 + 02 = 04
Practical: 4 Hrs./Week

Course Objectives

1) **Knowledge:** (i) Learn object oriented paradigm and its fundamentals.
2) **Skills:** (i) Understand Inheritance, Polymorphism and dynamic binding using OOP.
   (ii) Study the concepts of Exception Handling and file handling using C++ and Java.
3) **Attitude:** (i) Learn to apply advanced concepts to solve real world problems.

Course Outcomes

1) Apply the basic concepts of Object Oriented Programming to design an application.
2) Make use of Inheritance and Polymorphism to develop real world applications.
3) Apply the concepts of exceptions and file handling to store and retrieve the data.
4) Develop efficient application solutions using advanced concepts.

Laboratory Assignment 6

Demonstrate the use of Collection Frameworks in Java

Objective of the Assignment

- To study the concept of collection framework

Laboratory Assignment No: 6

Problem Statement

Write a Java program to

- create a new array list and print out the collection by position
- add some elements (string)
- search an element in an array list
- reverse elements in an array list
- test an array list is empty or not
- join two array lists

Introduction

- The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects.
- Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Contd..

- **What is Collection in Java**

  A Collection represents a single unit of objects, i.e., a group.

- **What is a framework in Java**

  It provides readymade architecture.

  It represents a set of classes and interfaces.

  It is optional.

- **What is Collection framework**

  The Collection framework represents a unified architecture for storing and manipulating a group of objects.

  It has:

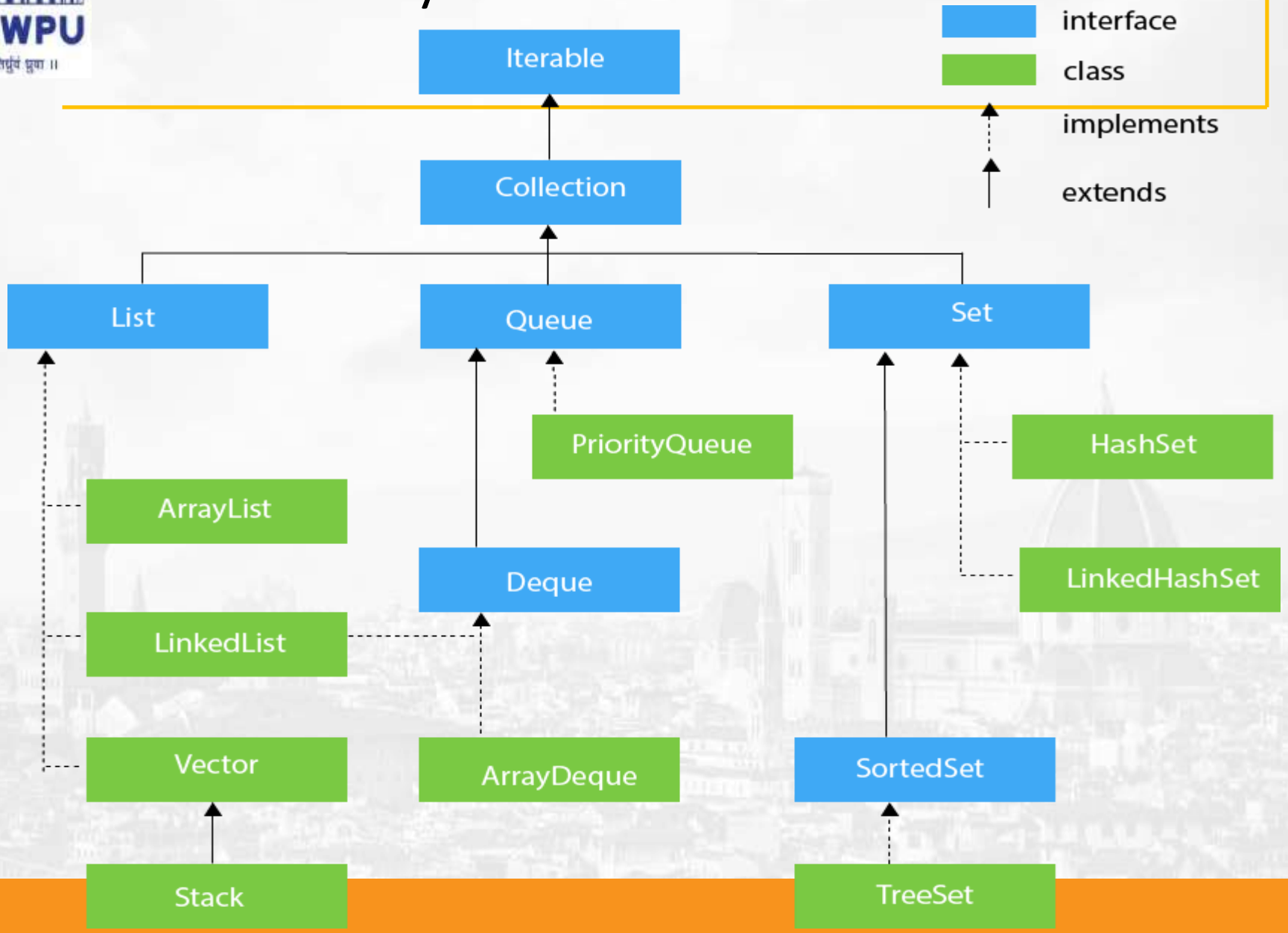
  Interfaces and its implementations, i.e., classes

  Algorithm

Generics

- Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type

Hierarchy of Collection Framework



Array List

- - The ArrayList class extends AbstractList and implements the List interface.

  - ArrayList supports dynamic arrays that can grow as needed.

- Standard Java arrays are of a fixed length.

  - After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.

  - Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged.

  -When objects are removed, the array may be shrunk.

Array List

- **ArrayList( )**

  This constructor builds an empty array list.

- **ArrayList(Collection c)**

  This constructor builds an array list that is initialized with the elements of the collection **c**.

- **ArrayList(int capacity)**

This 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.

Array List

- **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 out of range (index < 0 || index > size()).

- **boolean add(Object o)**

Appends the specified element to the end of this list.

- **void clear()**

Removes all of the elements from this list.

- **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.

Array List Example

```
import java.util.*;
 public class ArrayListDemo {
 public static void main(String args[]) {
                ArrayList al = new ArrayList(); // create an array list
                System.out.println("Initial size of al: " + al.size()); // add elements
                  al.add("C");
                  al.add("A");
                  al.add("E");
                  al.add("B");
                 al.add("D");
                 al.add("F");
                 al.add(1, "A2");
                 System.out.println("Size of al after additions: " + al.size()); // display
                 System.out.println("Contents of al: " + al); // Remove elements
                 al.remove("F");
                 al.remove(2);
                 System.out.println("Size of al after deletions: " + al.size());
                  System.out.println("Contents of al: " + al);
} }
```

Output

- 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]

Java - Data Structures

- The data structures provided by the Java utility package are very powerful and perform a wide range of functions. These data structures consist of the following interface and classes –
- Enumeration
- BitSet
- Vector
- Stack
- Dictionary
- Hashtable
- Properties
- All these classes are now legacy and Java-2 has introduced a new framework called Collections Framework

The Enumeration

- The Enumeration interface isn't itself a data structure. It defines a means to retrieve successive elements from a data structure.
- For example, Enumeration defines a method called nextElement that is used to get the next element in a data structure that contains multiple elements.

The Enumeration contd..

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

- **Object nextElement( )**

This returns the next object in the enumeration as a generic Object reference.

Example of Vector

```
import java.util.Vector;
 import java.util.Enumeration;
public class EnumerationTester {
 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

The BitSet Class

The BitSet

The BitSet class **creates a special type of array** that holds bit values. The BitSet array can increase in size as needed. This makes it similar to a vector of bits.

- The BitSet class implements a group of bits or flags that can be set and cleared individually.
- This class is very useful when you just assign a bit to each value and set or clear it as appropriate.
- **BitSet(int size)**
- This constructor allows you to specify its initial size, i.e., the number of bits that it can hold. All bits are initialized to zero.

- void **and**(BitSet bitSet) ANDs the contents of the invoking BitSet object with those specified by bitSet. The result is placed into the invoking object.

- **int cardinality( ):**Returns the number of set bits in the invoking object.

- **void clear( ):**Zeros all bits.

- **void clear(int index):** Zeros the bit specified by index.

- **void clear(int startIndex, int endIndex):**Zeros the bits from startIndex to endIndex.

- **void flip(int index):**Reverses the bit specified by the index.

- **void flip(int startIndex, int endIndex)**
  Reverses the bits from startIndex to endIndex.

Example of Bitset Class

```
import java.util.BitSet;
 public class BitSetDemo {
public static void main(String args[])
{
    BitSet bits1 = new BitSet(16);
    BitSet bits2 = new BitSet(16);
    // set some bits
    for(int i = 0; i < 16; i++)
        { if((i % 2) == 0) bits1.set(i);
         if((i % 5) != 0) bits2.set(i); }
    System.out.println("Initial
    pattern in bits1: ");
    System.out.println(bits1);
    System.out.println("\n Initial
    pattern in bits2: ");
        System.out.println(bits2); //
    AND bits
    bits2.and(bits1);
    System.out.println("\nbits2
        AND bits1: ");
     System.out.println(bits2); //
        OR bits
     bits2.or(bits1);
    System.out.println("\nbits2
        OR bits1: ");
     System.out.println(bits2); //
        XOR bits
     bits2.xor(bits1);
    System.out.println("\nbits2
        XOR bits1: ");
    System.out.println(bits2);
    }
    }
```

Output

- Initial pattern in bits1: {0, 2, 4, 6, 8, 10, 12, 14} Initial pattern in bits2: {1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}
- bits2 AND bits1: {2, 4, 6, 8, 12, 14}
- bits2 OR bits1: {0, 2, 4, 6, 8, 10, 12, 14}
- bits2 XOR bits1: {}

The Vector

- The Vector class is similar to a traditional Java array, except that it can grow as necessary to accommodate new elements.
- Like an array, elements of a Vector object can be accessed via an index into the vector.
- The nice thing about using the Vector class is that you don't have to worry about setting it to a specific size upon creation; it shrinks and grows automatically when necessary.

ArrayList	Vector
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity.	Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity.
3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.
5) ArrayList uses **Iterator** interface to traverse the elements.	Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also.

Example of Vector

```
import java.util.*;
public class VectorDemo {
public static void main(String args[]) { // initial size is 3, increment is 2 Vector
        v = new Vector(3, 2);
        System.out.println("Initial size: " + v.size()); System.out.println("Initial
    capacity: " + v.capacity()); v.addElement(new Integer(1));
        v.addElement(new Integer(2));
        v.addElement(new Integer(3));
        v.addElement(new Integer(4));
        System.out.println("Capacity after four additions: " + v.capacity());
        v.addElement(new Double(5.45));
         System.out.println("Current capacity: " + v.capacity());
        v.addElement(new Double(6.08));
        v.addElement(new Integer(7));
        System.out.println("Current capacity: " + v.capacity());
         v.addElement(new Float(9.4));
        v.addElement(new Integer(10));
```

```
System.out.println("Current capacity: " + v.capacity());
v.addElement(new Integer(11));
 v.addElement(new Integer(12));
System.out.println("First element: " +
(Integer)v.firstElement());
System.out.println("Last element: " +
(Integer)v.lastElement());
if(v.contains(new Integer(3)))
      System.out.println("Vector contains 3."); // enumerate
the elements in the vector.
Enumeration vEnum = v.elements();
System.out.println("\nElements in vector:");
while(vEnum.hasMoreElements())
 System.out.print(vEnum.nextElement() + " ");
 System.out.println(); }
}
```

output

- Initial size: 0 Initial capacity: 3
- Capacity after four additions: 5
- Current capacity: 5
- Current capacity: 7
- Current capacity: 9
- First element: 1
- Last element: 12
- Vector contains 3.
- Elements in vector: 1 2 3 4 5.45 6.08 7 9.4 10 11 12

The Stack

- The Stack class implements a last-in-first-out (LIFO) stack of elements.
- You can think of a stack literally as a vertical stack of objects; when you add a new element, it gets stacked on top of the others.
- When you pull an element off the stack, it comes off the top. In other words, the last element you added to the stack is the first one to come back off.

- **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 the 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.

Example of Stack

```
import java.util.*;
 public class StackDemo {
      static void showpush(Stack st, int a)
 {
      st.push(new Integer(a));
       System.out.println("push(" + a + ")"); System.out.println("stack: " + st); }
       static void showpop(Stack st)
 {
      System.out.print("pop -> ");
      Integer a = (Integer) st.pop();
       System.out.println(a);
      System.out.println("stack: " + st); }
public static void main(String args[])
       { Stack st = new Stack();
        System.out.println("stack: " + st);
      showpush(st, 42); showpush(st, 66); showpush(st, 99);
      showpop(st); showpop(st); showpop(st);
      try
       { showpop(st); }catch (EmptyStackException e)
              { System.out.println("empty stack");
 } } }
```

Output

- stack: [ ]
- push(42)
- stack: [42]
- push(66)
- stack: [42, 66]
- push(99)
- stack: [42, 66, 99]
- pop -> 99 stack: [42, 66]
- pop -> 66 stack: [42]
- pop -> 42
- stack: [ ]
- pop -> empty stack

The Dictionary

- The Dictionary class is an abstract class that defines a data structure for mapping keys to values.
- This is useful in cases where you want to be able to access data via a particular key rather than an integer index.
- Since the Dictionary class is abstract, it provides only the framework for a key-mapped data structure rather than a specific implementation.

- **Enumeration elements( )**

  Returns an enumeration of the values contained in the dictionary.
- **Object get(Object key)**

  Returns the object that contains the value associated with the key. If the key is not in the dictionary, a null object is returned.
- **boolean isEmpty( )**

  Returns true if the dictionary is empty, and returns false if it contains at least one key.
- **Enumeration keys( )**

  Returns an enumeration of the keys contained in the dictionary.
- **Object put(Object key, Object value)**

  Inserts a key and its value into the dictionary. Returns null if the key is not already in the dictionary; returns the previous value associated with the key if the key is already in the dictionary.
- **Object remove(Object key)**

  Removes the key and its value. Returns the value associated with the key. If the key is not in the dictionary, a null is returned.
- **int size( )**
- Returns the number of entries in the dictionary.

- The Dictionary class is obsolete. You should implement the **Map interface** to obtain key/value storage functionality.
- 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.

- **void clear( ):**Removes all key/value pairs from the invoking map.
- **boolean containsKey(Object k):**Returns true if the invoking map contains **k** as a key. Otherwise, returns false.
- **boolean containsValue(Object v):**Returns true if the map contains **v** as a value. Otherwise, returns false.
- **Set entrySet( ):**Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map.
- **boolean equals(Object obj):**Returns true if obj is a Map and contains the same 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 empty. Otherwise, returns false.

- **Set keySet( ):**Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
- **Object put(Object k, Object v):**Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
- **void putAll(Map m):**Puts all the entries from **m** into this map.
- **Object remove(Object k):**Removes the entry whose key equals **k**.
- **int size( ):**Returns the number of key/value pairs in the map.
- **Collection values( ):**Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Example

```
import java.util.*;
 public class CollectionsDemo {
 public static void main(String[] args) {
     Map m1 = new HashMap();
     m1.put("Zara", "8");
     m1.put("Mahnaz", "31");
     m1.put("Ayan", "12");
     m1.put("Daisy", "14");
     System.out.println();
      System.out.println(" Map Elements");
     System.out.print("\t" + m1); }
}
```

Output:-

Map Elements {Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}

The Hashtable

- The Hashtable class provides a means of organizing data based on some user-defined key structure.
- For example, in an address list hash table you could store and sort data based on a key such as ZIP code rather than on a person's name.
- The specific meaning of keys with regard to hash tables is totally dependent on the usage of the hash table and the data it contains.

Example of Hash Map

```
import java.util.*;
 public class HashTableDemo {
public static void main(String args[])
 { // Create a hash map
                Hashtable balance = new Hashtable();
                Enumeration names; String str; double bal;
                balance.put("Zara", new Double(3434.34));
                balance.put("Mahnaz", new Double(123.22));
                 balance.put("Ayan", new Double(1378.00));
                balance.put("Daisy", new Double(99.22));
                balance.put("Qadir", new Double(-19.08)); // Show all balances in hash table. names =
balance.keys();
                while(names.hasMoreElements())
                {
                str = (String) names.nextElement();
                System.out.println(str + ": " + balance.get(str)); }
                 System.out.println(); // Deposit 1,000 into Zara's account
                bal = ((Double)balance.get("Zara")).doubleValue();
                balance.put("Zara", new Double(bal + 1000));
                System.out.println("Zara's new balance: " + balance.get("Zara")); }
 }
```

OUTPUT:Qadir: -19.08 Zara: 3434.34 Mahnaz: 123.22 Daisy: 99.22 Ayan: 1378.0 Zara's new balance: 4434.34

The Properties

- Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.
- The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.
- The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

The **properties**

- The **properties** object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.
- It can be used to get property value based on the property key. The Properties class provides methods to get data and store data into properties file. Also used to get properties of system.
- *Advantage of properties file:*
- **Recompilation is not required, if information is changed from properties file:** If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.

- **String getProperty(String key)**
- Returns the value associated with the key. A null object is returned if the key is neither in the list nor in the default property list.
- **String getProperty(String key, String defaultProperty)**

Returns the value associated with the key; defaultProperty is returned if the key is neither in the list nor in the default property list.

- **void list(PrintStream streamOut)**

Sends the property list to the output stream linked to streamOut.

- **void list(PrintWriter streamOut)**

Sends the property list to the output stream linked to streamOut.

Example of Properties

```
import java.util.*;
public class PropDemo {
public static void main(String args[])
{
        Properties capitals = new Properties();
        Set states;
        String str;
        capitals.put("Illinois", "Springfield");
         capitals.put("Missouri", "Jefferson City");
        capitals.put("Washington", "Olympia");
        capitals.put("California", "Sacramento");
        capitals.put("Indiana", "Indianapolis"); // Show all states and capitals in hashtable.
         states = capitals.keySet(); // get set-view of keys
         Iterator itr = states.iterator();
        while(itr.hasNext())
        {   str = (String) itr.next();
        System.out.println("The capital of " + str + " is " + capitals.getProperty(str) + "."); }
        System.out.println(); // look for state not in list -- specify default
        str = capitals.getProperty("Florida", "Not Found");
        System.out.println("The capital of Florida is " + str + "."); } }
```

Interfaces (Set, List, Queue, and Dequeue)

- 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.

- **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( )**

  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.

```
import java.util.*;
public class SetDemo {
        public static void main(String args[])
{
         int count[] = {34, 22,10,60,30,22};
        Set<Integer> set = new HashSet<Integer>();
        try
        { for(int i = 0; i < 5; i++)
{
         set.add(count[i]); }
         System.out.println(set);
        TreeSet sortedSet = new TreeSet<Integer>(set);
        System.out.println("The sorted list is:");
        System.out.println(sortedSet);
        System.out.println("The First element of the set is: "+
        (Integer)sortedSet.first());
        System.out.println("The last element of the set is: "+
        (Integer)sortedSet.last()); }
        catch(Exception e) {} } }
```

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

Java LinkedList class

- Java LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.
- The important points about Java LinkedList are:
- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
- Java LinkedList class can be used as list, stack or queue.

Methods of Linked List

- **LinkedList()**It is used to construct an empty list.
- **LinkedList(Collection c):** It is used to construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.Methods of Java LinkedList
- **void add(int index, Object element):**It is used to insert the specified element at the specified position index in a list.void addFirst(Object o)It is used to insert the given element at the beginning of a list.
- **void addLast(Object o)**It is used to append the given element to the end of a list.int size()It is used to return the number of elements in a listboolean add(Object o)It is used to append the specified element to the end of a list.
- **boolean contains(Object o):**It is used to return true if the list contains a specified element.

- **boolean remove(Object o):**It is used to remove the first occurence of the specified element in a list.
- **Object getFirst():**It is used to return the first element in a list.
- **Object getLast():**It is used to return the last element in a list.
- **int indexOf(Object o):**It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
- **int lastIndexOf(Object o):**It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.

Example of Linked List

```
import java.util.*;
public class TestCollection7{
 public static void main(String args[])
{
        LinkedList<String> al=new LinkedList<String>();
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ravi");
        al.add("Ajay");

        Iterator<String> itr=al.iterator();
        while(itr.hasNext()){
        System.out.println(itr.next());
 }
 }
}
```

Output:Ravi Vijay Ravi Ajay

Difference between ArrayList and LinkedList

- Array List and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.
- ArrayList vs 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.

- **void add(int index,Object element):**It is used to insert element into the invoking list at the index passed in the index.
- **boolean addAll(int index,Collection c):**It is used to insert all elements of c into the invoking list at the index passed in the index.
- **object get(int index)**
- It is used to return the object stored at the specified index within the invoking collection.
- **object set(int index,Object element)**
- It is used to assign element to the location specified by index within the invoking list.
- **object remove(int index)**It is used to remove the element at position index from the invoking list and return the deleted element.
- **ListIterator listIterator()**It is used to return an iterator to the start of the invoking list.ListIterator
- **listIterator(int index)**It is used to return an iterator to the invoking list that begins at the specified index.

Example of Linked List

```
import java.util.*;
class TestArrayLinked{
 public static void main(String args[])
{
     List<String> al=new ArrayList<String>();//creating arraylist
     al.add("Ravi");//adding object in arraylist
     al.add("Vijay");
     al.add("Ravi");
    al.add("Ajay");
    List<String> al2=new LinkedList<String>();//creating linkedlist
    al2.add("James");//adding object in linkedlist
    al2.add("Serena");
    al2.add("Swati");
    al2.add("Junaid");
     System.out.println("arraylist: "+al);
 System.out.println("linkedlist: "+al2);
 }
}
```

Output

- arraylist: [Ravi,Vijay,Ravi,Ajay]
- linkedlist: [James,Serena,Swati,Junaid]

Classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, and TreeSet).

- Java Queue Interface
- Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.
- Queue Interface declaration
- **public interface** Queue<E> **extends** Collection<E>

Queue Methods

- boolean add(object):It is used to insert the specified element into this queue and return true upon success.
- boolean offer(object):It is used to insert the specified element into this queue.
- Object remove():It is used to retrieves and removes the head of this queue.
- Object poll():It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.
- Object element():It is used to retrieves, but does not remove, the head of this queue.
- Object peek():It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

PriorityQueue class

- The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.
- PriorityQueue class declaration
- The declaration for java.util.PriorityQueue class.
- **public class** PriorityQueue<E> **extends** AbstractQueue<E> **im plements** Serializable

```
import java.util.*;
class TestCollection12{
public static void main(String args[])
{
        PriorityQueue<String> queue=new PriorityQueue<String>();
        queue.add("Amit");
        queue.add("Vijay");
        queue.add("Karan");
        queue.add("Jai");
        queue.add("Rahul");
        System.out.println("head:"+queue.element());
        System.out.println("head:"+queue.peek());
        System.out.println("iterating the queue elements:");
        Iterator itr=queue.iterator();
        while(itr.hasNext())
        {
         System.out.println(itr.next());
        }
        queue.remove();
        queue.poll();
        System.out.println("after removing two elements:");
        Iterator<String> itr2=queue.iterator();
        while(itr2.hasNext()){
                        System.out.println(itr2.next());
        }
        }
}
```

Output

- head:Amit
- head:Amit
- iterating the queue elements: Amit Jai Karan Vijay Rahul
- after removing two elements: Karan Rahul Vijay

Hash Set

- HashSet Class: HashSet represents a set of elements (objects).
- It does not guarantee the order of elements. Also it does not allow the duplicate elements to be stored.
- HashSet class is represented as : class HashSet
- Object is created as : HashSet hs = new HashSet ();
- The following constructors are available in HashSet:
- HashSet();
- HashSet (int capacity); :capacity represents how many elements can be stored into the HashSet initially. This capacity may increase automatically when more number of elements is being stored.

Hash Class Method

Method	Description
boolean add(obj)	This method adds an element obj to the HashSet. It returns true if the element is added to the HashSet, else it returns false. If the same element is already available in the HashSet, then the present element is not added.
boolean remove(obj)	This method removes the element obj from the HashSet, if it is present. It returns true if the element is removed successfully otherwise false.
void clear()	This removes all the elements from the HashSet
boolean contains(obj)	This returns true if the HashSet contains the specified element obj.
boolean isEmpty()	This returns true if the HashSet contains no elements.
int size()	This returns the number of elements present in the HashSet.

program which shows the use of HashSet and Iterator

```
package com.myPack;
import java.util.HashSet;
import java.util.Iterator;
public class HashSetDemo {
public static void main(String[] args) {
//create a HashSet to store Strings
        HashSet <String> hs = new HashSet<String> ();
        //Store some String elements
        hs.add ("Anil");
        hs.add ("Akshara");
        hs.add ("Babji");
        hs.add ("Charan");
        hs.add ("Raman");
        // hs.add("India");
        //view the HashSet
        System.out.println ("HashSet = " + hs);
        //add an Iterator to hs
        Iterator<String> it = hs.iterator ();
        //display element by element using Iterator
        System.out.println ("Elements Using Iterator: ");
        while (it.hasNext() )
        {
                String s = (String) it.next ();
                System.out.println(s);  } }
}
```

Output

•

HashSet = [Akshara, Raman, Babji, Anil, Charan]

Elements Using Iterator:

Akshara

Raman

Babji

Anil

Charan

Sets: HashSet

```
HashSet<String> set = new HashSet<String>();
set.add("one");
set.add("two");
set.add("two");
set.add("two");
set.add("three");
System.out.println(set.size());
for (String s : set) {
        System.out.print(s + " ");
}
System.out.println(set.contains("two"));
```

```
3
one two three true
```

Tree based Implementation

- TreeSet

  iteration gives elements in sorted order
- To use a TreeSet,
  - either your objects must implement interface Comparable
  - or you must provide a Comparator object
- The elements in TreeSet stored in ascending order.
- TreeMap

  the keys are kept in a TreeSet
- To use a TreeMap
  - either your keys must implement interface Comparable
  - or you must provide a Comparator object for the keys
  - there is no requirement for the values

Example

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
} }
```

Output:

```
Ajay
Ravi
Vijay
```

Traversing List in Descending Order

```
import java.util.*;
class TreeSet2{
 public static void main(String args[])
{
                TreeSet<String> set=new TreeSet<String>();
                 set.add("Ravi");
                set.add("Vijay");
                set.add("Ajay");
                System.out.println("Traversing element through Iterator in descending order");
               Iterator i=set.descendingIterator();
               while(i.hasNext())
               {
                              System.out.println(i.next());
                }

               }
}
```

Output

Traversing element through Iterator in descending order

Vijay

Ravi

Ajay

Traversing element through NavigableSet in descending order

Vijay

Ravi

Ajay

Key learnings

- ArrayList Class
- Linked List Class
- Vector Class
- TreeSet Class
- Priority Queue Class

Thank You!!