UNIT - III

EXPLORING JAVA.LANG AND JAVA.UTIL

SYLLABUS [B.E - OU]

Exploring Java Language, Collections Overview, Collections Interfaces, Collection Classes, Iterators, Random Access Interface, Maps, Comparators, Arrays, Legacy Classes and Interfaces, String Tokenizer, BitSet, Date, Calendar, Timer.

OUTLINES

3.1 EXPLORING JAVA.LANGUAGE

The classes of java.lang package are fundamental and very essential to the Java application development. For this reason, this package is implicitly imported into source code even if the programmer does not import explicitly.

Important Interfaces of java.lang Package

(1) **CharSequence :** A CharSequence is a readable sequence of characters.

(2) **Cloneable :** A class implements the Cloneable interface to indicate to the Object that clone( ) is legal to be used by the class.

(3) **Comparable :** This interface gives ordering on the objects to the class that implements this interface.

(4) **Runnable :** This interface is used to create threads of a class.

Some interfaces like Appendable, Iterable and Readable are added from J2SE 5.

Important Classes of java.lang Package

(1) **Wrapper classes :** Every primitive data type has got corresponding wrapper class like, Boolean, Character, Double etc., where every data type can be viewed as an object.

(2) **Class :** A class object represents an object of a running Java application.

(3) **ClassLoader :** It is very often used by JVM to load a class into Java runtime environment.

(4) **Compiler :** It is used in support of JNI (Java Native Interface). This class provides a native code compiler.

(5) **Math :** The class Math provides many methods to do mathematical operations like basic exponential, logarithm, square root and trigonometric functions.

(6) **Object :** It is the root class of all classes of Java language.

(7) **Number :** The abstract class and superclass of wrapper classes like Integer, Double etc.

(8) **Process :** The methods of this class gives the capability to create a native process and return an instance of a subclass of Process. This object can be used by programmer to control the process and get the information of the process.

(9) **ProcessBuilder :** It provides a way to start and manage processes. Each process is an object of Process class.

(10) **StrictMath :** This class also contains the same methods of Math class. The methods of Math class may give different precision across many versions of JDK. This disadvantage is overcome in StrictMath class where precision is guaranteed.

(11) **System :** The System class contains many variables and methods useful in Java programming to interact with the operating system.

(12) **Thread :** Used to create threads of a class.

(13) **ThreadGroup :** A thread group represents a set of threads.

(14) **Throwable :** It is the superclass of all exceptions of Java.

(15) **Void :** The Void class is an uninstantiable class that can hold the reference to the class. Some classes like Enum, ProcessBuilder and StringBuilder are added in J2SE5.

3.1.1 Wrapper Classes

Java is a pure object oriented language as we can view everything in terms of an object. For example, a simple file abc.txt can be viewed as an object using File class, a system's address like www.yahoo.com using URL class and a data type int using Integer class.

Sometimes in Java, it is required, to convert all primitive data types as objects. For example, data structures of Java require objects only to store. That is, data structures cannot store data types directly. Data types must be converted into objects. Now comes wrapper classes. Wrapper classes are useful to convert data types into objects. As the name implies, wrapper class wraps the data type as an object.

There are eight wrapper classes for every data type defined in java.lang package. They are,

Table 3.1.1 Wrapper Classes to Primitive Types

Primitive	Wrapper
Boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short

Void is not a wrapper class, J2SE 5.0 provides this one to represent void return type. We cannot instantiate Void class. Void is used in java.lang.reflect package, to hold the reference of Class object.

```
int x = 10;
System.out.println(x);                    //prints 10
Integer i1 = new Integer(x);
System.out.println(i1);                   //prints 10
```

Now, i1 is an object of Integer class that wrapped (encapsulated) the data type x. Whereever, x is required as an object, as in data structures, we can use i1.

To get back the original data type int value, we use the intValue( ) method of Integer class that returns the data type int as follows,

```
int k = i1.intValue( );
```

3.1.1.1 Example Programs

EXAMPLE PROGRAM 1

Write a Java program that accepts numbers and converts it into string.

SOLUTION

```
import java.io.*;
class NumToStr
{
    public static void main(String args[ ]) throws IOException
    {
        BufferedReader buff = new BufferedReader(new InputStreamReader(System.in));
        String str;
        int num;
        System.out.println("Enter numbers or -999 to stop");
        do
        {
            num = buff.read( );              //reads a character as an integer
```

```
        try
        {
            str = Integer.toString(num);    //Converter an integer into string form
        }
        catch(NumberFormatException e)
        {
            System.out.println("Format is invalid");
            num = -999;
        }
        System.out.println("Number is" + num);
        System.out.println("Equivalent numeric string is" + str);
    } while(num!= -999);
  }
}
```

Output

```
Enter numbers or -999 to stop
```

EXAMPLE PROGRAM 2

Write a java program to find sum of a list of numbers entered by a user. It must convert the string representation of each number into an integer using the required string function.

SOLUTION

```
import java.io.*;
class Parsing
  {
    public static void main(String args[ ]) throws IOException
  {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  String s;
  int num, sum = 0;
  System.out.println("Enter the list of numbers, or 0 to quit");
```

```
        do
            {
            s = br.readLine( );                    //reads numbers as strings
            try
                {
                num = Integer.parseInt(s);  //converts string representation of number into
                                              integer

                }
                catch(NumberFormatException ex)
                    {
                    System.out.println("Incorrect Format");
                    num = 0;
                    }
                            sum = sum + nuk;
                } while(num! = 0)
                System.out.println("The sum is:" + sum);
            }
        }
```

Output

```
Enter numbers or 0 to quit
2
4
5
0
The sum is : 11
```

3.1.2 Number Class

The abstract class Number is the superclass of all numeric wrapper classes Byte, Double, Float, Integer, Long and Short. This abstract class provides many methods to convert wrapper objects into primitive data types.

Following is the class signature,

public abstract class number extends Object implements Serializable

The following are the methods that are defined in Number class which can be used by all its subclasses,

(1) **byte byteValue( ) :** Returns the value of the specified number as byte. This is not an abstract method.

(2) **short shortValue( ) :** Returns the value of the specified number as short. This is not an abstract method.

(3) **double doubleValue( ) :** Returns the value of the specified number as double. This is an abstract method.

(4) **float floatValue( ) :** Returns the value of the specified number as float. This is an abstract method.

(5) **long longValue( ) :** Returns the value of the specified number as long. This is an abstract method.

(6) **int intValue( ) :** Returns the value of the specified number as int. This is an abstract method.

Return values of all the above methods can be rounded to the nearest number.

3.1.3 Float and Double Classes

Java includes wrapper classes that represent (converts) primitive data types as objects. This conversion sometimes is required in Java to store data types in data structures like Stack, Vector, ArrayList, etc.

Float is the wrapper class for the data type float and Double is the wrapper class for double data type.

The following snippet of code illustrates the conversions.

(1) **Converting data types to wrapper objects using constructors**

(i)
```
int x = 10;
Integer i1 = new Integer(x);
System.out.println(i1);                    //prints 10 but 10 is in object form
```

(ii)
```
double y = 10.5;
Double d1 = new Double(y);
System.out.println(d1);                    //prints 10.5 but 10.5 is in object form
```

In the above code i1 and d1 represent data types x and y as objects respectively. Even though they print the values, they cannot be used in arithmetic calculations like addition and subtraction etc., because they are in object form.

(2) Converting wrapper objects into data types using wrapper methods

(i) `int k = i1.intValue( );`
`System.out.println(k);` //prints 10 and is in data type int form

(ii) `double m = d1.doubleValue( );`
`System.out.println(m);` //prints 10.5 and is in data type double form

Now, k and m can be used in arithmetic operations like addition or division etc., because they are primitive data types.

3.1.4 Parsing in Java

Sometimes in Java, it is required to convert string values into data types. For example, the user enters a value in a text field like marks, price etc., and the text field returns the value as a string. The same thing should be done in the case of command line arguments also. The process of converting strings into data types is called parsing that involves the special methods of type parseXXX( ) that are defined in wrapper classes (subclasses of Number class). There are eight wrapper classes namely boolean, byte, char, double, float, int, long and short, that define the methods like parseInt( ), parseDouble( ), parseFloat( ) etc.

Example

File Name : ParsingDemo.java

```
public class ParsingDemo
{
    public static void main(String args[ ])
    {
        String str1 = "10";
        System.out.println(str1);                    //prints 10 in string form
        int k = Integer.parseInt(str1);              //converting into data type
        System.out.println(k);                       //prints 10 in data type int form
        String str2 = "10.5";
        System.out.println(str2);                    //prints 10.5 but is in string form
        double m = Double.parseDouble(str2);         //converting into data type
        System.out.println(m);                       //prints 10.5 but is in data type double
                                                       form
    }
}
```

Output

10 10

10.5

10.5

3.1.5 Character Class

The Character class from java.lang package is a wrapper class for the data type char. It includes many methods used to validate user's input like user names and passwords.

Most Character class' methods are static that take at least one character (of type char) as argument. These methods perform either test (like is digit or not) or manipulate the character

Example : Changes to uppercase or to lowercase.

Methods

(1) **char charValue( ) :** Returns the character in a character object.

(2) **static boolean isDefined (char c) :** Returns true if the given character is defined in unicode and false otherwise.

(3) **static boolean isDigit(char c) :** Returns true if the given character is a digit and false otherwise.

(4) **static boolean isLetter(char c) :** Returns true if the given character is a letter and false otherwise.

(5) **static boolean isLowerCase(char c) :** Returns true if the given character is in lowercase and false otherwise.

(6) **static boolean isUpperCase(char c) :** Returns true if the given character is in uppercase and false otherwise.

(7) **static boolean isWhiteSpace(char c) :** Returns true if the given character is a white space and false otherwise.

(8) **static char toLowerCase(char c) :** Returns the lowercase character equivalent of given character.

(9) **static char toUpperCase( ) :** Returns the uppercase character equivalent of given character.

(10) **int compareTo :** Compares the invoking character object with the given character object and returns 0 if they are equal. Otherwise, it returns a negative value.

Example

File Name : CharacterDemo.java

```
public class CharacterDemo
{
    public static void main(String args[ ])
    {
        char c = 'A';
        System.out.println("is defined:" + Character.isDefined(c));
        System.out.println("is digit:" + Character.isDigit(c));
        System.out.println("is letter that can be used as the first letter of an identifier:"
        +Character.isJavaIdentifierStart(c));
        System.out.println("is a letter that can be used in the middle of identifier:" +
        Character.isJavaIdentifierPart(c));
        System.out.println("is letter:" + Character.isLetter(c));
        System.out.println("is letter of digit:" + Character.isLetterOrDigit(c));
        System.out.println("is lower case:" + Character.isLowerCase(c));
        System.out.println("is upper case:" + Character.isUpperCase(c));
        System.out.println("to upper case:" + Character.toUpperCase(c));
        System.out.println("to lower case:" + Character.toLowerCase(c));
        System.out.println("is space:" + Character.isWhitespace(c));
        }
    }
```

Output

```
is defined : true
is digit : false
is letter that can be used as the first letter of an identifier : true
is letter that can be used in the middle of identifier : true
is a letter : true
is letter or digit : true
is lower case : false
is upper case : true
to upper case : A
to lower case : a
is space : false
```

3.1.6 Object Class

Object class is the super class of all the classes in Java. So, every class created in Java extends the class Object by default. This means that a reference object of *Object* class can refer to an object of any other class. The reference object of *Object* class can also refer to an array since arrays in Java are treated as class.

Methods : *Object* class has the following methods, which are available in every object of any class.

(1) **Object clone( ) :** Method creates a duplicate copy of an object which is same as the object on which it is called. Only those classes that implements cloneable interface can be cloned. clone( ) is declared as protected in Object class.

(2) **boolean equals (Object obj) :** Method compares the contents of two objects to determine whether one object is equal to another. If both objects are equal, it returns true otherwise returns false. The equality of two objects vary depending on the type of objects being compared.

(3) **void finalize( ) :** Method is called before an unused object is destroyed by the Java run time system to free the resources used by that object.

(4) **int hashCode( ) :** Method returns the hash code associated with the object being called.

(5) **String toString :** Method returns a string that gives the description of the object on which it called. This is called automatically when an object is output using println( ) method. Most of the classes override this method.

(6) **void notify( ) :** Method starts the execution of a thread waiting on the involving object.

(7) **void notifyAll( ) :** Method starts the execution of all the threads that are waiting on the invoking object.

(8) **Class getClass( ) :** Method returns the class of an object at run time.

(9) **void wait( ) :** Method makes the current thread to sleep until some other thread starts executions. It has two other forms as follows.

```
void wait(long milliseconds)
void wait(long milliseconds, int nanoseconds).
```

Where the first form specifies waiting time in milliseconds and the second form specifies the time in nanoseconds as well.

The methods getClass( ), notify( ), notifyAll( ) and wait( ) are declared as final in Object class so they can not be overridden.

3.1.7 clone( ) and Cloneable Interface

The clone( ) method is defined by object class.

The clone( ) method makes a duplicate copy of an object. The classes which implement the Cloneable interface can only be cloned.

The Cloneable interface does not define any method. Its purpose is to indicate that a class implementing Cloneable interface can be cloned that is the copy of an object of that class can be made.

Cloning is not "enabled" by default in classes. If we call clone( ) on a class that does not implement Cloneable interface then a CloneNotSupportedException is thrown by the compiler. Cloning is considered to be important because Java's method calling semantics are called-by-reference, which allows the called method to modify the state of an object that is parsed to it. However, cloning can create several problems.

Let's consider an object which is being cloned contains a reference variable objref. Once the clone is made objref in the clone will refer to the same object. If modifications are made to objref then it will have its impact on the original object.

Consider Another Example : An object opens an I/O stream and has been cloned. Then, there will be two objects operating on the same stream. Let's say if one of the objects closes the stream, the other might still writing to it, thus causing an error. As cloning can cause problems the class java.lang. Object declares clone is method as protected. This means that only the class that implements Cloneable interface can call clone( ) method, of the class must explicitly override the clone( ) method to make it as public.

Example : The following example illustrates the use of clone( ) method and Cloneable interface.

```
class CloneClass implements Cloneable
{
    int x, y;
    CloneClass test( )
    {              //This method calls clone( ) in Object class
        try
        {
            return (CloneClass) super.clone( );
        }
        catch(CloneNotSupportedException e)
        {
```

```
                System.out.println("Exception" + e);
                return this;
            }
        }
    }
    class Demo
    {
        public static void main(String args[ ])
        {
            CloneClass obj1 = new CloneClass( );
            CloneClass obj2;
            obj1.x = 20;
            obj1.y = 30;
            obj2 = obj1.test( );                    //Clone obj1 as obj2
            System.out.println("obj1:" + obj1.x + " "+obj1.y);
            System.out.println("obj2:" + obj2.x + " "+obj2.y);
        }
    }
```

Output

```
obj1 : 20 30
obj2 : 20 30
```

3.1.8 Comparable Interface

Comparable interface allows the comparison of objects of a class. The class that implements Comparable interface contains objects can be compared. Comparable interface can be declared as,

```
interface Comparable <T>
```

Where, T represents the object type being compared.

Comparable interface declares a method called compareTo( ), which can be used for ordering the objects of the classes. The method is declared as follows,

```
int compareTo(T object)
```

This method compares the invoking object with object and returns zero, a negative value or a positive value.

zero If the values are equal.

Negative value - If the invoking object has a lower value.

Positive value - If the invoking object has a larger value.

In Java, several classes implemented the Comparable interface. They are Byte, Character, Double, Float, Short, Long, String and Integer. All these classes defines a compareTo( ) method.

Example

```
/*Program to illustrate Comparable interface */
    interface compare <T extends Comparable <T>>
    {
        T findminimum( );
        T findmaximum( );
    }
    class MinMax < T extends Comparable <T>> implements compare <T>
    {
        T[ ] num;
        MinMax(T[ ] obj)
        {
            num = obj;
        }
            public T findminimum( )
        {
            T n = num[0];
            for (int i = 1; i < num.length; i++)
            if(num [i].compareTo(n) < 0)
            {
                n = num [i];
                return n;
            }
            return null;
        }
```

```
    public T findmaximum( )
    {
        T n = num [0];
        for (int i = 1; i < num.length; i++)
        if(num[i].compareTo(n) > 0)
        {
            n = num[i];
            return n;
        }
        return null
    }
}
class CompareDemo
{
    public static void main(String agrs [ ])
    {
        Float fnums[ ] = {2.6f, 3.8f, 9.6f, 1.2f, 2.7f};
        Character ch[ ] = {'a', 'm', 'i', 's', 'j'};
        MinMax<Float> fobj = new MinMax<Float> (fnums);
        MinMax<Character> cobj = newMinMax<Character>(ch);
        System.out.println("The maximum value in float class:" +
        fobj.findmaximum( ));
        System.out.println("The minimum value in float class:" + fobj.findminimum(
        ));
        System.out.println("The maximum value in character class:" +
        cobj.findmaximum( ));
        System.out.println("The minimum value in character class:" +
        cobj.findminimum( ));
    }
}
```

3.1.9 Math Class

The Java class Math from java.lang package includes many useful methods and constants for arithmetic and mathematical manipulations. The Math class contains all static methods and constants. This means, it is not necessary to create an object of Math class and we can call the methods straight away.

The two constants of Math class are,

(1) Math.PI

(2) Math.E

They are defined in java.lang.Math class as,

public static final double E = 2.7182818284590452354;

public static final double PI = 3.14159265358979323846;

Example

In the following program, many methods of Math class are illustrated.

File Name : MathDemo.java

```
public class MathDemo
{
    public static void main(String args[ ])
    {
        System.out.println(Math.PI);                    //prints 3.141592653589793
        //abs( ) always returns a positive value
        System.out.println(Math.abs(-6.7));             //prints 6.7
        System.out.println(Math.abs(6.7));              //prints 6.7
        //ceil( ) always rounds to a higher value
        System.out.println(Math.ceil(6.7));             //prints 7.0
        System.out.println(Math.ceil(6.1));             //prints 7.0
        System.out.println(Math.ceil(-6.7));            //prints -6.0
        //floor( ) always rounds to a lower value
        System.out.println(Math.floor(6.7));            //prints 6.0
        System.out.println(Math.floor(6.1));            //prints 6.0
        System.out.println(Math.floor(-6.7));           //prints -7.0
        //round( ) always rounds in our normal way. > = 0.5 returns a higher value else
        lower value
```

```
        System.out.println(Math.round(6.7));          //prints 7
        System.out.println(Math.round(6.1));          //prints 6
        System.out.println(Math.round(-6.7));         //prints -6
        //max( ) returns the largest of the two numbers
        System.out.println(Math.max(6,7.5));          //prints 7.5
        //min( 0 returns the smallest of the two numbers
        System.out.println(Math.min(6,7.5));          //prints 6.0
        //pow( ) returns a value of first number raised to second number(power)
        System.out.println(Math.pow(2,3));            //prints 8
        //random( ) prints a random number lying between 0.0 and 1.0 (not inc      th)
        System.out.println(Math.random( ));
        //sqrt( ) returns the square root of the number
        System.out.println(Math.sqrt(25.5));          //prints 5.0497524
        //returns trigonometric values. The parameter for methods must be in radians
        System.out.println(Math.sin(20));             //prints 0.912945
        System.out.println(Math.cos(20));             //prints 0.40808
        System.out.println(Math.tan(20));             //prints 2.23716
    }
}
```

3.2 COLLECTIONS OVERVIEW

In Java programming, java.util package plays an important role as it includes many convenient classes like Random to generate random numbers, Date and Calendar classes to manipulate dates, StringTokenizer to tokenize strings into independent words and very important group of classes namely, data structures.

In Java, data structures had gone a metamorphic change between JDK versions.

(1) **JDK 1.0 :** Introduced Stack, Vector, Properties etc.

(2) **JDK 1.2 (J2SE 2) :** Introduced LinkedList, ArrayList, HashMap, HashTree etc., which as a group called as Collection framework.

(3) **JDK 1.5 (J2SE 5) :** Did not introduce any new data structures, but instead gave an easy manipulation methods to manipulate data structures like boxing/unboxing, generics and for each loop to easily iterate the elements of data structures.

The group of data structures introduced with JDK 2.0 is called Collections or Collection framework. Collection framework includes a hierarchy of interfaces and classes. A collection framework is a unique architecture for developing and manipulating collections. The superclass for all data structures is Collection interface.

The collection framework contains the following,

(1) **Interfaces :** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object oriented languages, interfaces generally form a hierarchy.

Examples : Collection, Map, List etc.

(2) **Implementations :** These are the concrete implementations of the collection interfaces. Mainly, they are reusable data structures.

Examples : LinkedList, ArrayList etc.

(3) **Algorithms :** These are the static methods that perform useful computations on collection classes, such as searching and sorting. The algorithms are said to be polymorphic; that is, the same method with different implementations can be used with different collection classes for different functionalities. In short, algorithms give reusable functionality. The algorithms can be compared with C++, STL (Standard Template Library) and Smalltalk collection hierarchy.

3.2.1 Advantages of the Java Collections Framework

(1) **Reduces Programming Effort :** By providing useful data structures and algorithms, the Collections Framework frees the programmer to concentrate on his programming logic. He need not write any extra code (like adapters) for interoperability.

(2) **Increases Program Speed and Quality :** The collections Framework includes data structures and algorithms that give high performance and high quality. The implementations are different for different classes. These implementations free the programmer from writing heavy code and there by programmer can concentrate on his logic and consequently productivity increases.

(3) **Allows Interoperability Among Unrelated APIs :** Collections Framework classes can operate with any kind of programming requirements. If the logic requires data as nodes or as column headings, Collection API provides.

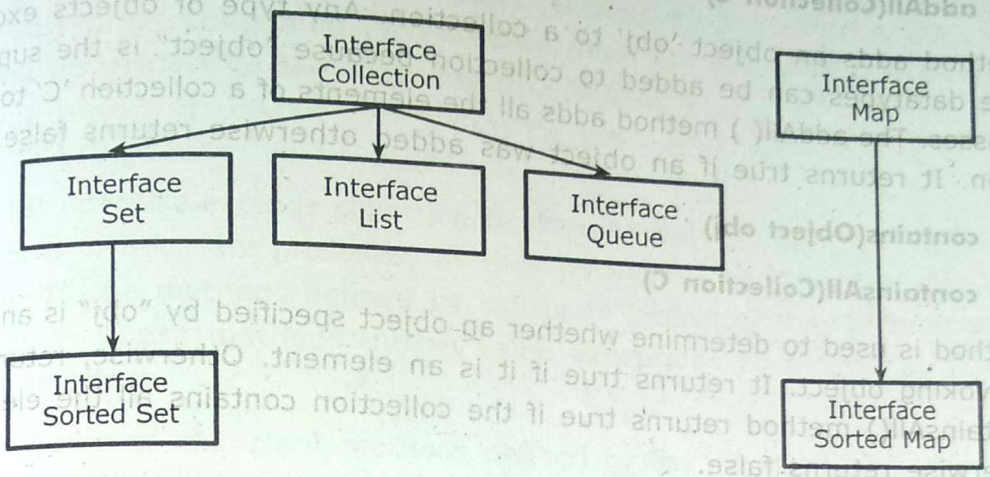
(4) **Reduces Effort to Learn and to Use New APIs :** Many Collection API classes look similar to any other API classes and interfaces. Collections Framework includes full fledged API classes for their functionalities without depending on other API sub frameworks.

(5) **Reduces Effort to Design New APIs :** Java programmers need not invent new classes to store data, but instead can use the Collection API classes.

(6) **Software Reusability :** The algorithms provide a log of reusability for different functionalities.

3.3 COLLECTIONS INTERFACES

All data structures are concrete classes derived from standard interfaces, which are shown in the Fig. 3.3.1. Core collection interfaces are the foundation of the Java collections framework.



(a) Does not Allow Duplicate Elements (b) Keys Must be Unique

Fig. 3.3.1 Hierarchy of Core Collection Interface

The above hierarchy contains two distinct trees, collection and map. Map is not a true collection because it is not a subclass of collection interface.

Interface name	Functionality
Collection	It is the root of collections hierarchy.
List	Stores a sequence of objects.
Queue	Stores objects but follows FIFO.
Set	Does not allow duplicate elements.
SortedSet	A special kind of Set, sorts the objects stored.
Map	Stores key/value pairs.
SortedMap	A special kind of Map, sorts the key objects stored.

Maps : Even though Map and SortedMap are not subclasses of collection interface, but they are part of collections framework.

3.3.1 Collection Interface

The collection interface is the parent of all the collections. It allows to work with groups of objects. It defines several methods that are available to all the collections. Some of these methods throws an UnSupportedOperationException if they are applied to unmodifiable collections. A ClassCastException is thrown if an incompatible object is added to a collection.

Methods : Some of the important methods are,

(1) **boolean add(Object obj)**

boolean addAll(Collection C)

This method adds an object 'obj' to a collection. Any type of objects except the primitive datatypes can be added to collection because "object" is the super class of all classes. The addAll( ) method adds all the elements of a collection 'C' to another collection. It returns true if an object was added otherwise returns false.

(2) **boolean contains(Object obj)**

boolean containsAll(Collection C)

This method is used to determine whether an object specified by "obj" is an element of an invoking object. It returns true if it is an element. Otherwise, returns false. The containsAll() method returns true if the collection contains all the elements of 'C'. Otherwise returns false.

(3) **boolean equals(Object obj) :** This method is used to compare two collections. It returns true if they are equal. Otherwise, returns false.

(4) **int size( ) :** This method returns the number of elements present within collection.

(5) **Object[ ] toArray( ) :** This method returns all the elements stored within a collection in an array.

3.3.2 Set and Sortedset Interfaces

Set Interface : The set interface extends collection interface and defines a set. It does not allow the duplicate elements within a set. If an existing elements is added to the set using add( ) method then the method returns false. This interface does not define its own methods.

SortedSet Interface : The SortedSet interface extends the sort interface to define a sorted set. The set is sorted in ascending order. It defines the methods in addition to methods defined by Set Interface. These methods throws a NoSuchElementException if the set is empty. ClassCastException if the objects are in compatible and NullPointerException if null object is used when null object is not allowed in the set.

Methods : The methods defined by SortedSet interface are,

(1) **Object first( ) :** This method is used to get the first element present in the sorted set.

(2) **Object last( ) :** This method is used to get the last element present in the sorted set.

(3) **SortedSet headSet(Object endObj) :** This method returns the subset of the sorted set containing elements starting from the beginning upto the object specified by endObj.

(4) **SortedSet tailSet(Object startObj)** : This method returns to subset of the sorted set containing the elements starting from the object specified by startObj upto the end of the set.

(5) **SortedSet subSet(Object startObj, Object endObj)** : This method returns the subset of the sorted set containing elements between startObj and endObj. The invoking object also refers to the elements of a subset.

3.3.3 List Interface

The list interface extends collection to store a sequence of elements (list of objects). The elements of a list are processed by their positions. The list allows duplicate element. In addition to the methods defined by collection interface, the list interface defines its own methods. These methods throws same exceptions as thrown by the methods of Collection interface.

Methods : Some of important methods defined by list interface are,

(1) **void add(int index, Object obj)**

boolean addAll(int index, Collection C)

The add( ) method adds an element into a list at the specified position "index". The addAll( ) method adds all the elements of a collection 'C' into the list at the specified position "index". If there is element(s) at that position or beyond that insertion point then they are shifted up. Thus, these methods do not overwrite elements. The addAll( ) method returns true if the changes were made else it returns false.

(2) **Object get(int index)**

Object set(int index, Object obj)

The get( ) method returns the object stored at the specified index within the list. The set( ) method changes a value of an element specified by index within the list.

(3) **int indexOf(Object obj)**

int lastIndexOf(Object obj)

The indexOf( ) method returns the index of the first occurrence of the 'obj' within the list. The lastindexOf( ) method returns the index of the last occurence of the 'obj' within the list. Both methods returns –1 if 'obj' does not belong to the invoking list.

(4) **List subList(int startIndex, int endIndex) :** This method returns the sublist of a list. The length of sublist to be extracted is specified by the startIndex and endIndex.

(5) **Object remove(int index) :** This method deletes an element specified by index from the list. The method removes the deleted element.

3.4 COLLECTION CLASSES

The subclasses of collection interface are called collection classes.

Some of the collection classes provide full implementation to collection interfaces so they can be used as it is, and others are abstract classes. The collection classes defined by collection framework are,

(1) AbstractCollection.

(2) AbstractList.

(3) AbstractSequentialList.

(4) LinkedList.

(5) ArrayList.

(6) AbstractSet.

(7) HashSet.

(8) LinkedHashSet.

(9) TreeSet.

Few of the collection classes are discussed below.

Concrete Collection Classes

(1) **ArrayList Class :** The ArrayList class inherits the functionalities of AbstractList class and List interface. The ArrayList creates the dynamic arrays. The size of an array need not to be mentioned while creating the array list. The size of an array list grows dynamically to accommodate new objects and shrinks when object are removed.

Constructors

ArrayList( )

ArrayList(Collection coll)

ArrayList(int size)

The first form of constructors creates an empty list. The second constructor creates and initializes an array list with the elements of the Collection 'coll'. The third constructor specifies the initial size of arraylist which can grow dynamically when elements are added to it.

(i) **void ensureCapacity(int size) :** This method increases the size of an array list to the size specified by 'size'.

(ii) **void trimToSize( ) :** This method reduces the size of an array list.

Example

```
import java.util.*;
class ColorsList
{
    public static void main(String args[ ])
    {
        ArrayList color = new ArrayList( );
        System.out.println("Initial size of colorlist:" + color.size( ));
        color.add("red");
        color.add("green");
        color.add("magenta");
        color.add(0,"orange");
        System.out.println("Size of color list after addition:" + color.size( ));
        System.out.println("Color list is:" + color);
        color.remove("red");
        System.out.println("size of color list after deletion:" + color.size( ));
        System.out.println("color list is:" + color);
    }
}
```

Output

```
Initial size of color list : 0
Size of color list after addition : 4
Color list is : [orange, red, green, magenta]
size of color list after deletion : 3
color list is : [orange, green, magenta]
```

(2) **LinkedList Class :** The LinkedList class inherits AbstractSequentialList class and List interface. It creates a linked list.

Constructors

LinkedList( )

LinkedList(Collection coll)

The first constructor creates an empty linked list while the second constructor initializes the linked list with elements of the Collection 'coll'

Methods

(i) void addFirst(Object obj)

void addLast(Object obj)

The addfirst( ) and addlast( ) method inserts an object 'obj' at the beginning and at the end of the linked list respectively.

(ii) Object getFirst( )

Object getLast( )

The getFirst( ) method returns the first element and getLast( ) method returns the last element of the linked list.

(iii) Object removeFirst( )

Object removeLast( )

The removeFirst( ) method deletes the first element and removeLast( ) method deletes the last element from the linked list.

Example

```
import java.util.*;
class LListDemo
{
    public static void main(String args[ ])
    {
        LinkedList color = new LinkedList( );
        color.add("cyan");
        color.add("white");
        color.addFirst("Yellow");
        color.add(2, "pink");
```

```
        color.addLast("blue");
        color.add("light green");
        System.out.println("Linked list after addition:" + color);
        color.remove(1);
        color.removeFirst( );
        color.removeLast( );
        System.out.println("Linked list after deletion:" + color);
        Object obj = color.get(2);
        System.out.println("The color at index 2:" + (String) obj);
        color.set(2, "red");
        System.out.println("Linked list after setting:" + color);
    }
}
```

Output

```
Linked list after addition : [Yellow, cyan, pink, white, blue, light green]
Linked list after deletion : [pink, white, blue]
The color at index 2 : blue
Linked list after setting : [pink, white, red]
```

(3) **HashSet Class :** The HashSet inherits AbstractSet class and Set interface. The HashSet class is used to store the collection created by it into the hash table and the hash table stores the unique key. These keys are converted into hash code by applying a hashing technique. This hash code is then used to retrieve the data associated with the key. The hash table does not store the elements in any sorted order. Hashing is useful to perform the operations such as add( ), remove( ), size() and contains( ).

Constructors

```
HashSet( )
HashSet(Collection coll)
HashSet(int size)
HashSet(int size, float fill_ratio)
```

The first form of constructor creates an empty hash set. The second form of constructor initializes the hash set with the elements of collection 'coll'. The third form of constructor specifies the size of the hash set. The last form of constructor specifies both the size and the fill ratio. The fill ratio specifies the allowed capacity of hash set that can be full before it is resized. It ranges between 0.0 and 1.0. The default value for fill ratio is 0.75.

Example

```
import java.util.*;
class Hset
{
    public static void main(String args[ ])
    {
    HashSet color = new HashSet( );
    color.add("white");
        color.add("green");
        color.add("black");
        color.add("blue");
        System.out.println("The hash set is :" + color);
    }
}
```

Output

The hash set is : [green, white, blue, black]

(4) LinkedHashSet Class : The LinkedHashSet class inherits HashSet class. It creates linked list for the hash set. The order of elements stored in linked list is same in which the elements were inserted.

This is useful to process the LinkedHashSet using an iterator because it returns the elements in the order in which they were inserted. This class does not define its own methods.

Constructors

LinkedHashSet( )

LinkedHashSet(Collection coll)

LinkedHashSet(int initialcap)

LinkedHashSet(int initiallCap, float load-factor)

The first form of constructor creates an empty linked has set with the default initial capacity (i.e., 16) and load factor (i.e., 0.75). The second form of constructor initializes the linked hash set with the elements of collection 'coll'. The third form of constructor specifies the initial capacity. The last constructor specifies both the initial capacity and the load factor.

Example

```
import java.util.*;
class LHset
{
    public static void main(String args[ ])
    {
        LinkedHashSet color = new LinkedHashSet( );
        color.add("white");
        color.add("green");
        color.add("black");
        color.add("blue");
        System.out.println("The linked hash set is :" + color);
    }
}
```

Output

The linked hash set is : [white, green, black, blue].

(5) **TreeSet Class :** The TreeSet class inherits Set interface. It creates a collection that uses a tree for storage. Unlike the HashSet class, TreeSet stores the objects in stored order i.e., in ascending order. Therefore the large amount of sorted information can be accessed and retrieved very fast.

Constructor

TreeSet( )
TreeSet(Collection coll)
TreeSet(Comparator cp)
TreeSet(SortedSet sort)

The first form of constructor creates a default tree set whose elements will be sorted in ascending order. The second constructor initializes the tree set with the elements of the Collection 'coll'. The third constructor creates an empty tree set whose elements will be sorted in the order specified by the Comparator object 'cp'. The final constructor initializes the tree set with the elements of the SortedSet 'sort'.

Example

```
import java.util.*;
class TSet
{
    public static void main(String args[ ])
    {
        TreeSet color = new TreeSet( );
        color.add("brown");
        color.add("white");
        color.add("yellow");
        color.add("cyan");
        System.out.println("The tree set is:" + color);
    }
}
```

Output

```
The tree set is : [brown, cyan, white, yellow]
```

3.5 COLLECTION ALGORITHMS

One of the components of the collections framework is algorithms (the others are interfaces and implementation class). The algorithms are pieces of reusable functionality provided by the Java collections. Algorithms can be used with collection classes and map classes. The *Collection class contains many static methods, which are commonly known as algorithms.* These static methods can be applied to collections classes for data manipulations.

JDK 1.5 (J2SE 5) added many methods (like checked methods) to the already existing algorithms, especially generics are added. Generics give type safe elements in a data structure. That is, if we set the generic to strings, we can add only strings to a data structure.

The static methods take first argument as the collection class on which the operation is to be performed. The great majority of the algorithms provided by the Java platform operate on List instances, but a few of them operate on other collection classes.

The algorithms are helpful for the following operations on collections.

(1) Sorting.

(2) Shuffling.

(3) Routine Data Manipulation.

(4) Searching.

(5) Composition.

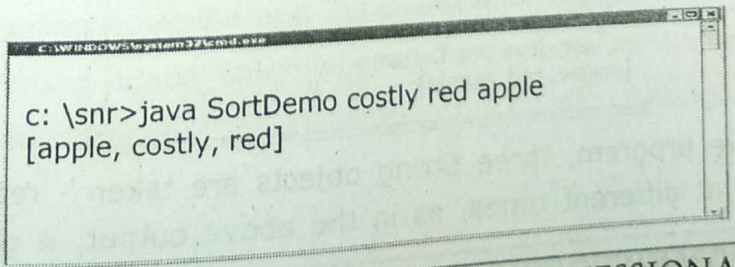
(6) Finding Extreme Values.

(1) **Sorting :** The sort algorithm reorders a List so that its elements are in ascending order.

Example : The following algorithm sorts the list passed as command line argument to the program

```
import java.util.*;
public class SortDemo
{
    public static void main(String[ ] args)
        {
            List list = Arrays.asList(args);
            Collections.sort(list);
            System.out.println(list);
        }
}
```

Output



(2) **Shuffling :** The shuffle algorithm does the reverse of what sort does. That is, the elements are never sorted but instead shuffled. If executed at different times, different outputs are obtained; that is shuffling will change. It orders the elements randomly at different times. It takes all possible permutations. This algorithm is useful in implementing games of chance.

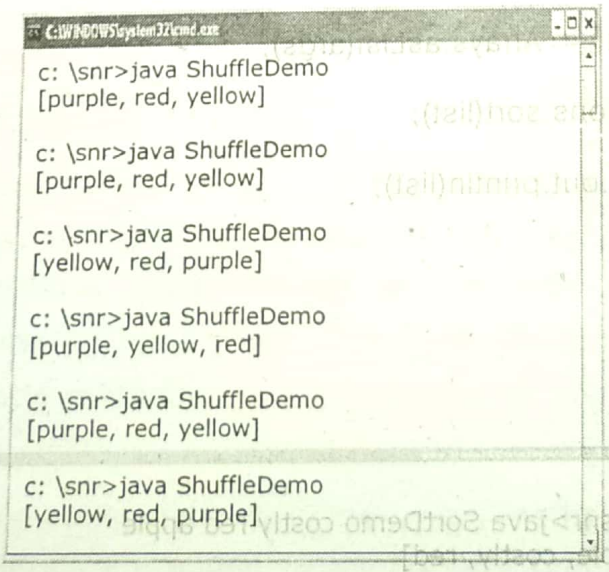
Example : It could be used to shuffle a list of playing cards representing a deck. Also, it's useful for generating test cases while testing software.

Example

File Name : ShuffleDemo.java

```
import java.util.*;
public class ShuffleDemo
{
    public static void main(String[ ] args)
    {
        List list = new ArrayList( );
        list.add("red");
        list.add("purple");
        list.add("yellow");
        Collections.shuffle(list);
    System.out.println(list);
    }
}
```

Output



In the above program, three String objects are taken - red, purple and yellow. When executed at different times, as in the above output, it gives different orders.

(3) **Routine Data Manipulation :** The Collections class provides five algorithms for performing routine data manipulation on List objects. They are,

(i) **reverse :** Reverse the order of the elements in a list.

(ii) **fill :** Overwrites every element in a List with the specified value. This operation is useful for reinitializing a List.

(iii) **copy :** Takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, overwriting its contents. The destination. List must be at least as long as the source. It it is longer, the remaining elements in the destination List are unaffected.

(iv) **swap :** Swaps the elements at the specified positions in a List.

(v) **addAll :** Adds all the specified elements to a Collection. The elements to be added may be specified individually or as an array.

(vi) **min :** Finds the minimum element in a list.

(vii) **max :** Finds the maximum element in a list.

Example

In the following program, reverse of the elements and minimum and maximum operations are done.

File Name: MinMaxReverse.java

```
import java.util.*;
public class MinMaxReverse
{
    public static void main(String[ ] args)
    {
        List list = new ArrayList( );
        list.add(new Integer(10));
        list.add(new Integer(20));
        list.add(new Integer(30));                    //reverse operation
        System.out.println("Original elements: "+ list);
        Collections.reverse(list);
        System.out.println("After reversing: " + list);
        //retrieving minimum and maximum of elements
        System.out.println("\nMinimum is "+ Collections.min(list));
        System.out.println("Maximum is " + Collections.max(list));
    }
}
```

Output

Original elements : [10, 20, 30]

After reversing : [30, 20, 10]

Minimum is 10

Maximum is 30

3.6 ITERATORS

Iterator : In computer vocabulary, iterator means an object with which we can travel all through a range of elements. In Java, Iterator is an interface that allows programmers to access a collection.

Following is the signature of iterator interface.

public interface Iterator : Enumeration interface is replaced by Iterator in collections frame work. The advantages of iterator over Enumeration are,

(1) Iterator allow the caller to remove elements from the underlying collection during the iteration with well defined semantics.

(2) Method names have been improved.

(3) We can print from any index number, but need not all the elements.

It includes three abstract methods as follows,

boolean hasNext( ) : Returns true if the collection has more elements.

Object next( ) : Returns the next element in the collection.

void remove( ) : Removes the current element in the collection.

In order to use iterator the following steps should be followed.

STEP 1 : Obtain an object of Iterator on the collection class by calling iterator( ) method on the collection. (by default, the pointer is placed on the first element so that all the elements are printed. But, we can be choosy on the number of elements to print).

STEP 2 : Create a control loop to iterate through all the elements by using hasNext( ) method.

STEP 3 : Extract each element by calling next( ) method

The following code illustrates these steps.

```
Iterator it = list.Iterator( );                    //1st step
while(it.hasNext( ));                              //2nd step
{
    System.out.println(it.next( ));                //3rd step
}
```

Here list is an object of a collection (e.g : ArrayList).

ListIterator : ListIterator is an interface that allows programmers to loop through the collection in both forward and backward directions. It also allows, to modify the element in collection. It is used only for those collections that implement List. It has the following methods.

(1) **void add(Object o) :** The add( ) method inserts the given element before the element returned by next( ).

(2) **boolean hasNext( ) :** The hasNext( ) method returns true if there is an element after the current element in the collection. Otherwise, it returns false.

(3) **hasPrevious( ) :** The hasPrevious( ) method returns true if there is an element before the current element. Otherwise, it returns false.

(4) **Object next( ) :** The next( ) method returns the element after the current element, it there is one. Otherwise, it throws NoSuchElementException.

(5) **int nextIndex( ) :** The nextIndex( ) method returns the index of the next element. However, if there is no next element, it returns the size of the list.

(6) **Object previous( ) :** The previous( ) method returns the element before the current element. If there is no previous element, it throws NoSuchElementException.

(7) **int PreviousIndex( ) :** The previousIndex( ) method returns the index of the previous element, if there is one. However if there is no previous element, it returns -1.

(8) **void remove( ) :** The remove( ) method deletes the current element from the list. It throws IllegalStaticException if there is no current element.

(9) **void set(Object o) :** The set( ) method assigns the given object to the current element in the list.

The following code illustrates how to use ListIterator interface,

```
ListIterator li = list.listIterator( );
while (li.hasNext( ))
{
    System.out.println(li.next( ));
}
```

Here, list is on object of a collection (**Example :** ArrayList). The listIterator( ) method returns on object of ListIterator.

3.7 RANDOM ACCESS INTERFACE

RandomAccess is added to Collection framework with JDK 1.4 version. RandomAccess is called a marker interface because it does not contain any methods or variables. The other marker interfaces are Serializable and Cloneable. Every marker interface has some functionality or capability.

The RandomAccess interface helps to have faster random access to the list elements. RandomAccess must be used with only List because it is the List which implements RandomAccess interface in the total collection framework. That is, the subclasses of List like ArrayList and LinkedList etc., can use the RandomAccess interface.

Following snippet of code gives how to use RandomAccess interface,

```
LinkedList list = new LinkedList( );
Object obj;
if(list instanceof RandomAccess)
{
    for(int i = 0; i < list.size( ); i++)
    {
        obj = list.get(i);
        System.out.println (obj);
    }
}
```

It is used for faster access and thereby for performance tuning.

3.8 MAPS

A map is an object that stores key/value pairs. Each key is unique and has its associated value, which may be repeated. Some maps allow keys and values to be null. Both key and value are object. Maps allow to find a value for a given key. An important point about maps is that, they cannot obtain an iterator to cycle through a map. Thus, for-each style for loop cannot be used with maps.

There are four map interfaces,

(1) Map.

(2) SortedMap.

(3) NavigableMap.

(4) MapEntry.

(1) **Map Interface :** A map stores key/values pairs. Map does not allow duplicate keys. Each key maps to one value. When we supply the key, Map returns the value associated with it. A few maps allow null values and null keys also.

Following is the class signature,

public interface Map

Map and its subclasses cannot use Iterator to retrieve the elements because Map does not implement. Iterator interface. Keys must be unique. That is, map does not allow duplicate keys. Keys and values must be object of any Java class.

Methods : The following are the methods defined by the map interface,

(i) **void clear( ) :** Deletes all key/value pairs of the invoking Map object.

(ii) **boolean containsKey(Object key) :** Returns true if the given key is found in the invoking map object. Otherwise, returns false.

(iii) **boolean containsValue(Object value) :** Returns true if the given value is found in the invoking Map object. Otherwise, returns false.

(iv) **Set<Map.Entry<key, value>> entrySet( ) :** Returns a set of (key, value) pairs in the invoking Map object in the form of Map.Entry objects.

(v) **boolean equals(Object o) :** Returns true if the given object is a Map object and has same (key, value) pairs as those in the invoking Map object. Otherwise, returns false.

(vi) **V get(Object key) :** Returns the value corresponding to the given key. Return null if the key is not found.

(vii) **int hashCode( ) :** Returns the hash code for the invoking Map object.

(viii) **boolean isEmpty( ) :** Returns true if the invoking Map object is empty. Otherwise, returns false.

(ix) **Set<K> keySet( ) :** Returns a set of keys that are present in the invoking Map object.

(x) **V put<K key, V value) :** Inserts the given key/value pair in the invokingMap object. If the key already exist then the put( ) overwrites the value with the given value and returns the overwritten value. However, if the key is not present then it returns a null.

(xi) **Void putAll(Map<? extends K, ? extends V>map) :** Inserts the key/value pairs of the given map object into the invoking Map object.

(xii) **V remove(Object key) :** Removes that key, value pair whose key matches with the given key.

(xiii) **int size( ) :** Returns the number of elements in the invoking Map object.

(xiv) **Collection<V> values( ) :** Returns a collection of values present in the invoking Map object.

Example : The example Program 3.8.1 illustrate the use of Map interface.

```
import java.util.*;
class MapExample
{
    public static void main(String args[ ])
    {
        Map<Object, String>map = new HashMap<Object, String>( );
        map.put(new Integer(3), "Three");
        map.put(new Integer(4), "Four");
        map.put(new Integer(5), "Five");
        map.put(new Integer(1), "One");
        map.put(new Integer(2), "Two");
        System.out.println("Number of elements in map :" + map.size( ));
        System.out.println("Key/value pairs are :");
        System.out.println(map);
    }
}
```

Output

```
Number of elements in map : 5
Key/value pairs are : [3 = Three, 4 = Four, 5 = Five, 1 = One, 2 = Two]
```

Program 3.8.1 Example of Map Interface

(2) SortedMap Interface : It is just a sorted version of Map where elements can be sorted. It extends Map interface. A SortedMap is a Map that maintains its entries in ascending order which are sorted according to the key's natural ordering. We can specify the ordering also by supplying a Comparator.

Following is the class signature,

```
public interface SortedMap extends Map
```

Methods : The following are the methods defined by the sortedMap interface,

(i) **Comparator<? super K> comparator( ) :** Returns the comparator of the invoking SortedMap object. Returns null if natural ordering is used for the invoking Map object.

(ii) **K firstKey( ) :** Returns the first key in the invoking Map object.

(iii) **K lastKey( ) :** Returns the last key in the invoking Map object.

(iv) **SortedMap<K, V> subMap(K from, K to) :** Returns a SortedMap object that contains a set of key/value pairs whose keys are greater than or equal to 'from' and less than 'to'.

(v) **SortedMap<K, V> tailMap(k from) :** Returns a SortedMap object that contains a set of key/value pairs whose keys greater than or equal to 'from'.

(vi) **SortedMap<K, V> headMap(K to) :** Returns a SortedMap object that contains a set of key/value pairs whose keys less than 'to'.

Example : The Program 3.8.2 uses the two methods of SortedMaps-firstkey( ) and lastkey().

```
import java.util.*;
public class SortedMapDemo
{
public static void main(String args[ ])
{
    SortedMap smap=new TreeMap( );
    //Insert some (key, value) pairs in smap
    smap.put("7", "Seven");
    smap.put("25", "Twenty Five");
    smap.put("9", "Nine");
    smap.put("1", "One");
System.out.println("The lowest key in the sortedmap is : " +smap.firstkey( ));
System.out.println("The highest key value in the sorted map is : " +smap.lastkey( ));
System.out.println("The key values in the sortedmap in increasing order are : "+smap);
}
}
```

Output

```
The lowest key in the sortedmap is : 1
The highest key in the sortedmap is : 25
The key values in sortedmap in increasing order are :
{1 = One, 7 = Seven, 9 = Nine, 25 = Twenty Five}
```

Program 3.8.2 Example of SortedMap Interface

(3) **NavigableMap Interface :** The NaVigableMap interface extends SortedMap. It supports the retrieval of (key, value) pairs that closely match to the given key or keys.

Methods : The following are the methods defined by NavigableMap interface,

(i) **Map.Entry<K, V> ceilingEntry(K key) :** Searches the Map object for the smallest key, which is greater than or equal to the given key. Returns the entry for the smallest key, if it is found. Otherwise, returns null.

(ii) **K ceilingKey(K key) :** Searches the Map object for the smallest key, which is greater than or equal to the given key. Returns the entry for the smallest key, if it is found. Otherwise, returns null.

(iii) **NavigableSet<K> descendingKeySet( ) :** Returns a NavigableSet whose keys are in decreasing order of the keys in invoking Map object.

(iv) **NavigableMap<K, V> descendingMap( ) :** Returns a NavigableMap whose keys and values are in decreasing order of the keys and values in invoking Map object.

(v) **Map.Entry<K, V) firstEntry( ) :** Returns the first(key, value) pair whose key is the smallest one in the map.

(vi) **K higherKey(K Key) :** Searches the set for the largest key that is greater than the given key. Returns the largest key if it is found. Otherwise, returns null.

(vii) **Map.Entry<K, V> lastEntry( ) :** Returns the entry that has the largest key in the map.

(viii) **NavigableSet<K> navigableKeySet( ) :** Returns a NavigableSet that has the keys of the invoking map.

(ix) **Map.Entry<K, V> pollFirstEntry( ) :** Returns the first entry that has the smallest key and removes it from the map. Returns null if the map is empty.

(x) **Map.Entry<K, V> pollLastEntry( ) :** Returns the last entry that has the largest key and removes it from the map. Returns null if the map is empty.

Example : The Program 3.8.3 uses some of the methods of NavigableMap interface.

```
import java.util.*;
import java.util.concurrent.*;
public class NavigableMapDemo
{
public static void main(String args[ ])
{
    NavigableMap<Integer, String> nmap=new TreeMap,<Integer, String>( );
    //Insert some elements
    nmap.put(1, "Summer");
    nmap.put(2, "Winter");
    nmap.put(3, "Rainy");
    nmap.put(4, "Autumn");
    nmap.put(5, "Fall");
    System.out.println("The Navigable map in descending order is :" +
    nmap.descendingMap());
    System.out.println("The first entry in the Navigable map is :" +nmap.firstEntry( ));
    System.out.println("The last entry in the Navigable map is :" + nmap.lastEntry( ));
    System.out.println("Removing the first entry :" + namp.pollFirstEntry( ));
    System.out.println("Removing the last entry :" + nmap.pollLastEntry( );
    System.out.println("The Navigable map in descending order is :" +
    nmap.descendingMap());
    }
}
```

Output

```
The navigable map in descending order is :
{5 = Fall, 4 = Autumn, 3 = Rainy, 2 = Winter, 1 = Summer}
The first entry in the Navigable map is : 1 = Summer
The last entry in the Navigable map is : 5 = Fall
Removing the first entry : 1 = Summer
Removing the last entry : 5 = Fall
The navigable map in descending order is,
{4 = Autumn, 3 = Rainy, 2 = Winter}
```

Program 3.8.3 Example of NavigableMap Interface

(4) **Map.Entry Interface :** The Map.Entry interface is an inner class of Map that allows programmers to work with each entry in a map.

Methods : The following are the methods defined by the Map.Entry interface,

(i) **boolean equals(Object object) :** Returns true if the given object is a Map.Entry object whose key and value matches with the key and value of the invoking object.

(ii) **K getKey( ) :** Returns the key of the invoking Map.Entry object.

(iii) **V getValue( ) :** Returns the value of the invoking Map.Entry object

(iv) **int hashCode( ) :** Returns the hash code for the invoking Map.Entry object.

(v) **V setValue(V value) :** Sets the value of the invoking Map.Entry object to the given value. It throws ClassCastException if the given value is not the correct type for the invoking Map object and IllegalArgumentException is thrown if there is any problem with the given value. A NullPointerException is thrown if the given value is null and the invoking Map object restricts null values. It throws UnsupportedOperationException if the invoking Map object cannot be changed.

Example : The Program 3.8.4 uses some of the methods of Map.Entry Interface.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Iterator;
import java.util.Set;
public class MapEntryDemo
{
    public static void main(String args[ ])
    {
        Map map=new HashMap( );
        //Insert some elements in the map
        map.put("7", "Sunday");
        map.put("3", "Wednesday");
        map.put("1", "Monday");
        map.put("2", "Tuesday");
        map.put("4", "Thursday");
        map.put("6", "Saturday");
```

```
        map.put("5", "Friday");
        Set s=map.entrySet( );
        Iterator i=s.iterator( );
        System.out.println("The key/value pairs are :");
        while(i.hasNext( ))
        {
            Map.Entry e=(Map.Entry) i.next( );
            System.out.println(e.getKey( )+" " + e.getValue( ));
        }
    }
}
```

Output

```
The key/value pairs are :
1  Monday
2  Tuesday
3  Wednesday
4  Thursday
5  Friday
6  Saturday
7  Sunday
```

Program 3.8.4 Example of Map.Entry Interface

3.9 COMPARATORS

In general, a comparator compares two items of data. Java also means the same. In Java, Comparator is used to compare two stored objects in a Set of a Map etc.

The java.util.Comparator interface is used to dictate in which order we want the sorting of the objects of a data structure. Generally, Comparator is not needed if there's a natural sorting order. Natural sorting means alphabetical order - A before B or 1 before 2 etc. A TreeSet of a TreeMap, by default, gives a natural sorting order, but if we want a different order, then we require comparator.

We pass a comparator object, which defines the sorting order, as a parameter to a TreeMap or TreeSet constructor. A Comparator interface defines two methods compare( ) and equals( ).

```
public int compare(Object obj1, Object obj2)
public boolean equals(Object obj).
```

(1) **compare( ) :** Method takes two Objects and returs 0, 1, -1 depending upon the parameters passed. 0 os returned when both objects are same. If first object is greater than second, returns 1 else -1.

(2) **equals( ) :** Method returns boolean value true, if both objects are same. Otherwise, it returns false. Generally, we do not override this method because an implemented concrete method is inherited from Object class.

3.9.1 Example Programs

EXAMPLE PROGRAM 1

Using comparator write a Java program to sort a tree set in a reverse order.

SOLUTION

```
import java.util.*;
//Implement Comparator interface
class MyComparator implements Comparator<String>
{
    //Override compare( ) method to reverse the order
        public int compare(String str1, String str2)
    {
        String s1, s2;
        s1 = str1;
        s2 = str2;
    //compareTo( ) invoked by s2 reverses the tree set
    return s2.compareTo(s1);
    }
}
class RevTreeSet
{
    public static void main(String args[ ])
    {
        //Create a tree set of strings
        TreeSet<String> tree = new TreeSet<String>(new MyComparator( ));
        //Add elements
```

```
        tree.add("Ashraf");
        tree.add("Shazia");
        tree.add("Ameer");
        tree.add("Rubin");
        tree.add("Haseena");
        //Display the treeSet
        System.out.println("TreeSet in a reverse order is");
        for(String element : tree)                    //Display the elements
        System.out.println(element);
    }
}
```

Output

```
Shazia
Rubin
Haseena
Ashraf
Ameer
```

EXAMPLE PROGRAM 2

Write Java program using a comparator to sort accounts by last name.

SOLUTION

```
import java.util.*;
class Acompare implements Comparator<String>
{
    public int compare(String s1, String s2)
    {
        int a, b, c;
        String string1, string2;
        string1 = s1;
        string2 = s2;
```

```
            a = string1.lastIndexOf(' ');
            b = string2.lastIndexOf(' ');
            c = string1.substring(a).compareTo(string2.substring(b));
            if(c ==0)
            return string1.compareTo(string2);
            else
            return c;
        }
    }
    class AccountComparing
    {
        public static void main(String args[ ])
        {
            TreeMap<String, Double> treemap = new
            TreeMap<String, Double>(new Acompare( ));
            treemap.put("Rafael Nadal", new Double(1497.09));
            treemap.put("Roger Fedrer", new Double(1001.02));
            treemap.put("Sarena Williams", new Double(206.03));
            treemap.put("Maria Williams", new Double(-153.06));
            Set<Map.Entry<String, Double>> s=treemap.entrySet( );
            for(Map.Entry<String, Double>mapentry:s)
            {
                System.out.print(mapentry.getKey( ) +":");
                System.out.println(mapentry.getValue( ));
            }
        }
    }
```

Output

```
Roger Fedrer : 1001.02
Rafael Nadal : 1497.09
Maria Williams : -153.06
Sarena Williams : 206.03.
```

3.10 ARRAYS

Arrarys is a class in java.util package introduced from JDK 1.5 version, which provides many static methods to do operations on arrays like filling, comparing, sorting and searching. Earlier to the introduction of Arrays, these operations were done from scratch. Now, the coding has become easier with Arrays. Many static methods are overloaded.

Following is the class signature:

```
public class Arrays extends Object
```

Methods : Following are some static methods with their description.

(1) **static int binarySearch(double myarray[ ], double num) :** It is used to find a specified number (num) in the array (myarray) using binary search. This method should be used with sorted arrays. This method is overloaded that can take a byte array, char array, int array etc.

(2) **static void equals (double myarray1[ ], double myarray2[ ]) :** It is used to check whether two arrays have the same elements. This method is overloaded that can take a byte array, char array, int array etc.

(3) **static void fill(double myarray1[ ], double num) :** It is used to fill all the array elements of the array (myarray1) with the specified value (num). This method is overlaoded that can take a byte array, char array, int arry etc.

(4) **static void sort(double myarray1[ ]) :** It is used to sort the elements in the ascending order. This method is overloaded that can take a byte array, char array, int array etc.

Example : The following program illustrates the use of Arrays class.

```
import java.util.*;
public class ArraysMethods
{
    public static void main(Strings[ ] args)
    {
        int marks[ ] = {50, 30, 20, 10, 40};                    //integer array
        int price[ ] = {50, 30, 20, 10, 40};
        //to compare two arrays whether their elements are same
        System.out.println("Both arrays are equal: "+ Arrays.equals(marks, price));
        System.out.print("Original values: ");
        for(int i = 0; i < marks.length; i++)
        {
            System.out.print(marks[i]+" ");
        }
        System.out.println( );      //sorting the elements of the array in ascending order
        Arrays.sort(marks);         //elements in the original array itself are changed
```

```
        System.out.print("Values After sorting: ");
        for(int i = 0; i < marks.length; i++)
        {
            System.out.print(marks[i]+" ");
        }
        int k = Arrays.binarySearch(marks, 30); //to find the index number of an element
        System.out.println("\nIndex number of 30 in the sorted array: "+k);
    }
}
```

Output

```
Both arrays are equal : true
Original values : 50 30 20 10 40
Values after sorting : 10 20 30 40 50
Index number of 30 in the sorted array : 2
```

3.11 LEGACY CLASSES AND INTERFACES

Legacy means synchronization. The data structures introduced with JDK 1.0 version are called legacy classes and interfaces. Eventhough, the Collections framework is far more superior with its rich set of features, the legacy classes are not depreciated. Instead, some more methods are added and reengineered to fit into collection framework.

Table 3.11.1 gives a list of legacy classes and interfaces.

Table 3.11.1 Legacy Collection Classes

Legacy Structure	Type of the Class	Functionality
Enumeration	interface	Used to iterate and print the elements
Vector	class(non-abstract)	Stores all types of objects
Stack	class(non-abstract)	Stores all types of objects(LIFO order)
Dictionary	abstract class	Stores key/value pairs
Hashtable	class(non-abstract)	Stores key/value pairs
Properties	class(non-abstract)	Stores key/value pairs

3.11.1 Vector

Vector class is an expandable array of objects. It can grow at runtime as and when the elements are added to it, to accommodate the new elements. When elements are removed it shrinks automatically. It is like ArrayList, but with one difference. Vector methods are synchronized where as ArrayList methods are not synchronozied. For this reason, performance-wise ArrayList is better. With legacy vector, we use Enumeration interface instead of Iterator. But in J2SE 5, Vector is reengineered (re-modelled) to fit into collections framework and now Vector can use generics and iterator.

Following is the class signature of Vector class (as in JDK 1.5),

public class Vector extends AbstractList implements List, RandomAccess, Cloneable, Serializable

Constructors : The following are the constructors of Vector,

(1) Vector( ).

(2) Vector(int size).

(3) Vector(int size, int incr).

(4) Vector(Collection coll).

The first constructor creates a vector with the default size (10). The second constructor specifies the size of the vector. The third constructor specifies size as well as the increment, which indicates the amount of extra space to be allocated during each reallocation. The fourth constructor creates a vector from the elements of specified collection.

Methods : The following are the methods of Vector,

(1) **void addElement(Object obj) :** Adds an element to the Vector.

(2) **void insertElementAt(Object obj, int index) :** Inserts the obj at the specified index. The first element gets 0 by default.

(3) **Object firstElement( ) :** Returns the first element stored in the Vector.

(4) **Object lastElement( ) :** Returns the last element stored in the Vector.

(5) **Object elementAt(int index) :** Returns the element at the specified index.

(6) **int indexOf(Object obj) :** Returns the index number of the specified element in the Vector.

(7) **boolean contains(Object obj) :** Returns true if the specified element exists.

(8) **int size( ) :** Returns the number of elements present in the Vector.

(9) **int capacity( ) :** Returns the storing capacity of the Vector.

(10) **void trimToSize( ) :** Deletes the extra capacity.

(11) **Enumeration elements( ) :** Returns an object of Enumeration interface.

Example : The Program 3.11.1 illustrates the methods of Vector.

File Name : VectorTest.java

```
import java.util.*;
public class VectorTest
{
    public static void main(String args[ ])
    {
        Vector vect = new Vector( );      // creates vector with default capacity 10
        vect.addElement(new Integer(5));
        vect.addElement(new Float(5.7F));
        vect.addElement(new String("Hello"));
        vect.addElement("Universal");
        Double d = new Double(15.76);
        vect.addElement(d);
        String str = "World";
        vect.insertElementAt(str, 1);
        System.out.println("First element :" + vect.firstElement( ));
        System.out.println("Last element :" + vect.lastElement( ));
        System.out.println("4th element in vector :" + vect.elementAt(3));
        System.out.println("Index of Hello :" + vect.indexOf("Hello"));
        System.out.println("Element Universal exists :" + vect.contains("Universal"));
        System.out.println("Size of vector :" + vect.size( ));
        System.out.println("Capacity of vector before trimming :" + vect.capacity( ));
        vect.trimToSize( );
        System.out.println("Capacity of vector after trimming :" + vect.capacity( ));
        System.out.println(vect);                    // prints all elements in a line
        Enumeration e = vect.elements( );            // assign the elements to
                                                     Enumeration to print
```

```
        while(e.hasMoreElements( ))
        {
            System.out.println(e.nextElement( ));
        }
        String s = (String) vect.elementAt(1);
        System.out.println("str is :" + s);
    }
}
```

Output

```
C:\pub>java VectorTest
First element : 5
Last element : 15.76
4th element in vector : Hello
Index of Hello : 3
Element Universal exists : true
Size of vector : 6
Capacity of vector before trimming : 10
Capacity of vector after trimming : 6
[5, World, 15.7, Hello, Universal, 15.76]
5
World
15.7
Hello
Universal
15.76
str is : World
```

Program 3.11.1 Using Vector

3.11.2 HashTable

HashTable is a concrete implementation of the dictionary class. It also stores information in the form of key value pairs but in a hash table. To store a pair, first a hash code is determined. After which, the hash code is used as an index and the values are stored in the hash table at that index.

Constructors : The constructors of the HashTable are,

(1) **Hashtable( ) :** It creates a hash table with default size of 11.

(2) **Hashtable(int size) :** It creates a Hash table with the specified size.

(3) **Hashtable(int size, float fillRatio) :** It creates a hash table with the specified size and fill ratio between 0.0 and 1.0.

(4) **Hashtable(Map m) :** It creates a Hash table with the elements in map m.

Hashtable was part of original java.util package. However, with the introduction of collections, Hashtable was made to implement Dictionary as well as map interfaces. Thus, Hashtable has the methods of both of these interfaces.

Methods : Following are the legacy methods defined by Hashtable,

(1) **void clear( ) :** The clear( ) method first resets the hash table and then empties it.

(2) **Object clone( ) :** The clone( ) method duplicates the invoking object and returns the same.

(3) **boolean contains(Object value)**

boolean containsValue (Object value) : These methods return true if there is some vlaue in the hashtable that matches the given value. Otherwise, they return false.

(4) **boolean containsKey(Object key) :** The containsKey( ) method returns true if there is some key in the hash table that matches the given key. Otherwise, it returns false.

(5) **Enumeration <V> elements( ) :** The elements( ) method returns an enumeration of values in the hash table.

(6) **V get(Object Key) :** The get( ) method returns the object that has the value corresponding to the given key. It returns a null object, if the key is not found in the hash table.

(7) **boolean isEmpty( ) :** The isEmpty( ) method returns true if the hash table is empty. Otherwise, it returns false.

(8) **Enumeration <K> keys( ) :** The keys( ) method returns an enumeration of the keys in the hash table.

(9) **V put(K key, V value) :** The put( ) method inserts the given key and value in the hash table. Then, it returns the previous value of the key if the key is already in the hash table. It returns a null object, if the key is not in the hash table.

(10) **void rehash( ) :** The rehash( ) method first increases the size of the hash table and then rehashes all the keys in the hash table.

(11) **V remove(Object key) :** The remove( ) method deletes the given key and its corresponding value from the hash table and returns the value. It returns a null object, if the key is not in the hash table.

(12) **int size( ) :** The size( ) method returns the number of key/value pairs in the hash table.

(13) **String toString( ) :** The toString( ) method returns the string representation of the hash table.

Example : Hashtable methods are illustrated in the Program 3.11.2.

File Name : HashDemo.java

```
import java.util.*;
public class HashDemo
{
    public static void main(String args[ ])
    {
        Hashtable ht = new Hashtable( );
        System.out.println("Before adding elements :" + ht.isEmpty( ));
        Integer i1 = new Integer(5);
        ht.put("hello", i1);                    //insert a key/value pair
        System.out.println("After adding elements :" + ht.isEmpty( ));
        ht.put("World", new Integer(10));
        ht.put("Apple", new Double(20.5));
        ht.put("Banana", new Boolean(true));
        ht.put("World", new Integer(10));
        HashDemo hd1 = new HashDemo( );
        ht.put(hd1, i1);
        ht.put("Universal", "Kothi");
        ht.put("Date", "12th March, 2001");
        StringBuffer sb = new StringBuffer( );
        ht.put(sb, "Config Software Solutions");
```

```
        System.out.println("Before removing size :" + ht.size( ));
        ht.remove("Apple");
        System.out.println("After removing size :" + ht.size( ));
        System.out.println("Hello object in ht :" + ht.containsKey("Hello"));
        System.out.println("i1 object in ht :" + ht.contains(i1));
        System.out.println("sb is associated with :" + ht.get(sb));
        Enumeration e = ht.keys( );
        while(e.hasMoreElements( ))
        {
            Object k = e.nextElement( );
            Object v = ht.get(k);
            System.out.println("Key :" + k + "Value :" + v);
        }
    }
}
```

Output

```
C:\pub>javaC HashDemo.java
C:\pub>java HashDemo
Before adding elements : true
After adding elements : false
Before removing size : 8
After removing size : 7
Hello object in ht : true
i1 object in ht : true
sb is associated with : Config Software Solutions
Key : date Value : 12th March, 2001
Key : Value : Config Software Solutions
Key : HashDemo@256a7c        Value : 5
Key : Hello  Value : 5
Key : Banana     Value : true
Key : World  Value : 10
```

Program 3.11.2 Using Hashtable

3.11.3 Stack

Stack is a well-known data structure with LIFO(Last-In-First-Out) pattern which means that the last item added to the stack is the first one to be removed. It is a subclass of vector. It has the following constructor, Stack( ). It creates an empty stack.

Example : The Program 3.11.3 illustrates stack operations like push( ), pop( ) etc.

File Name : StackDemo.java

```
import java.util.*;
public class StackDemo
{
    public static void main(String args[ ])
    {
        Stack st = new Stack( );
        System.out.println(st.empty( ));  // prints true
        st.push(new Integer(5));
        st.push(new Boolean(true));
        st.push(new Float(5.56F));
        st.push(new Double(59.76));
        System.out.println("Poping the element :" + st.pop( ));
                                    // pops the last element added, that is 59.76
        System.out.println("Stack is empty before while loop :" + st.empty( ));
                                    // returns false as stack is not empty
        while(! st.empty( ))        // iterates until all elements are popped out
        {
            System.out.println(st.pop( ));
        }
        System.out.println("Stack is empty after while loop :" + st.empty( ));
                                    // prints true as all the elements are popped out
    }
}
```

Output

```
C:\pub>java StackDemo
true
Poping the element : 59.76
Stack is empty before while loop : false
5.56
true
5
Stack is empty after while loop : true
```

Program 3.11.3 Using Stack

3.11.4 Dictionary (Abstract)

Dictionary is an abstract class in the java.util package and operates much like Map interface. Like Map, it stores the elements as a list of key value pairs. It has the following methods,

(1) **elements( ) :** The element( ) method returns an enumeration of values in the dictionary. Its syntax is,

Enumeration <V> elements( )

where, V is the type of values.

(2) **get( ) :** The get( ) method returns the object that has the value of the given key. It returns null object if the key is not found in the dictionary. Its syntax is,

V get(Object key)

(3) **isEmpty( ) :** The isEmpty( ) method returns true if the dictionary is empty. Otherwise, it returns false. Its syntax is,

boolean isEmpty( )

(4) **keys( ) :** The keys( ) method returns an enumeration of keys in the dictionary. Its syntax is,

Enumeration <K> keys( )

where, K is the type of the keys.

(5) **put( ) :** The put( ) method inserts a key and its corresponding value in the dictionary. It returns the precious value of the key if the key is already in the dictionary. It returns null, if the key is not in the dictionary. Its syntax is,

V put(K key, V value)

(6) **remove( ) :** The remove( ) method deletes a key and its corresponding value from the dictionary and returns the deleted value. It returns null if the key is not in the dictionary. Its syntax is,

```
V remove(Object key)
```

(7) **size( ) :** The size( ) method returns the number of (key, value) pairs in the dictionary. Its syntax is,

```
int size( )
```

From the above methods it is obvious that, a dictionary object is much like a Map or List object.

3.11.5 Enumeration Interface

The Enumeration interface provides a standard means of iterating through a list of sequentially stored elements in a data structure (this interface must be implemented by the data structure). It is replaced with Iterator interface in collections framework.

Methods : It provides two abstract methods using which iteration can be done,

(1) **hasMoreElements( ) :** Returns false when no elements exists.

(2) **nextElement( ) :** Retrieves the next element in the enumeration.

Method signature of the above methods defined in the java.util package,

```
public abstract boolean hasMoreElements( );
public abstract java lang.Object.nextElement( );
```

Following snippet of code illustrates how iteration can be done,

```
Enumeration e = vect.elements( );
while(e.hasMoreElements( ))
{
    System.out.println(e.nextElement( ));
}
```

elements( ) method of Vector class returns an object of Enumeration interface. hasMoreElements( ) returns true as long as elements are in the Enumeration object. If all elements are exhausted it returns false and the loop terminates. nextElement( ) returns each object stored in the Enumeration interface object.

3.11.6 Properties

Properties is a subclass of Hashtable class. It maintains a list of values in which both keys and values are represented as String objects. It has an instance variable defaults, which holds a default list of each property object.

Constructors : The following are the constructors of Properties,

(1) **Properties( ) :** It creates a Properties object without any default values.

(2) **Properties(Properties defaultValues) :** It creates a Properties object with the given default values.

Methods : Properties class inherits all methods of HashTable. In addition to HashTable methods, it defines the following methods,

(1) **String getProperty(String key)**

String getProperty(String key, String defaultValue)

These methods return the value of the given key. However, the first method returns null if the given key is not present either in the default property list or in the list. Whereas, the second method returns the given default value if the key is not available in both the lists.

(2) **Void List (PrintStream outStream)**

Void List (PrintWriter outStream)

These methods send the property list to the output stream that is connected to the outStream.

(3) **Void load(InputStream inStream)**

Void load (Reader inStream)

These methods take a property list as input from the input stream that is connected to the inStream. These methods throw an IOException on error.

(4) **Void loadFromXML(InputStream inputStream)**

This method takes a property list as input from an XML document that is connected to the inStream. This method throws IOException and InvalidPropertiesFormatException on errors.

(5) **Enumeration <?> PropertyNames( ) :** This method returns an enumeration of the keys present in list and default property list.

(6) **Object setProperty(String key, String value) :** This method associates the given value with the given key and returns the previous value of the key, if the key is already present. However, it returns null if the key is not present.

(7) **Void store(OutputStream outStream, String descp)**

Void store(OutputStream outStream, String descp)

These methods first write the given string description and then write the property list to the output stream that is connected to outStream. These methods throw an IOException on error.

(8) **Void StoreToXML(OutputStream OutStream, String descp) :** This method first writes the given string description and then writes the property list to the XML document that is connected to the outStream. This method throws an IOException on error.

(9) **Set<String> stringPropertyNames( ) :** This method returns a set of keys.

Example

File Name : PropetiesDemo.java

```
import java.util.*;
public class PropertiesDemo
{
    public static void main(String args[ ])
    {
        Properties p = new Properties( );
        p.put("Bawarchi", "Biryani");
        p.put("Blue Sea", "Tea");
        p.put("Niagara", "Irani Haleem");
        System.out.println(p.getProperty("Blue Sea"));
        System.out.println(p.getProperty("Hotel Adab"));
        String str;
        Enumeration e = p.keys( );
        while(e.hasMoreElements( ))
        {
            str = (String) e.nextElement( );
            System.out.println(str + " is famous for " + p.getProperty(str));
        }
        String str1 = p.getProperty("Hotel Adab", "Kaju Roti");
        System.out.println("Hotel Adab is famous for " + str1);
    }
}
```

Output

```
Tea
null
Niagara is famous for Irani Haleem
Bawarchi is famous for Biryani
Blue Sea is famous for Tea
Hotel Adab is famous for Kaju Roti
```

Program 3.114 Using Properties

3.12 STRING TOKENIZER

In programming languages, statements are divided into individual pieces like identifiers, keywords and operators etc., called tokens. Java provides StringTokenizer class that breaks a string into its component tokens. This class is available in java.util package. Tokens are separated from one another by delimiters (special operators), white space characters such as blank, tab, new line and carriage return.

3.12.1 String Tokenizer Constructors

(1) StringTokenizer(String str)

(2) StringTokenizer(String str, String delimiters)

(3) StringTokenizer(String str, String delimiters, boolean delimiterAsToken)

In all constructors str is a string that is tokenized.

In the first constructor, the default delimiter is used.

In the second and third constructors delimiters in string specifies the delimiters.

In the third constructor if 'delimiterAsToken' is true then the delimiters are also returned as tokens when the string is tokenized otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two constructors.

3.12.2 String Tokenizer Methods

(1) **int countTokens( ) :** Using the current set of delimiters the method determines the number of tokens left to be tokenized and returns the result.

(2) **boolean hasMoreTokens( ) :** It returns true if one or more tokens remain in the string and returns false if there are none.

(3) **boolean hasMoreElements( ) :** This method determines whether there are one or more tokens available in the string to extract. It returns true if there are one or more tokens, otherwise it returns false.

(4) **object nextElement( ) :** This method is used to obtain the next token. The token is returned as an object.

(5) **String nextToken( ) :** It returns the next token as string.

(6) **String nextToken(String delims) :** This method is used to obtain the next token as a string and set the delimiters string.

Example : To use StringTokenizer class to tokenize a string.

File Name : StringTokenizerDemo.java

```
import java.util.*;

public class StringTokenizerDemo

{

    public static void main(String args[ ])

    {

        String str = "All of us wish India to win the World Cup";

        StringTokenizer st = new StringTokenizer(str);

        System.out.println("No. of tokens in the string : " + st.countTokens( ));

                                                    //prints 10

        System.out.println("Following are the tokens : ");

        while(st.hasMoreTokens( ))

        {

            System.out.println(st.nextToken( ));

            //print each word in a separate line

        }

    }

}
```

countTokens( ) methods returns the number of tokens (words) present in the string. hasMoreTokens( ) method returns true as long as tokens exists with the StringTokenizer object, st. When no tokens exists to display, the method returns false and thereby the while loop terminates, nextToken( ) returns the next token (word) available with the StringTokenizer object st.

Output

```
No. of tokens in the string : 10
Following are the tokens :
All
of
us
Wish
India
to
Win
the
World
Cup
```

Program 3.12.1 Using StringTokenizer Class

```
//Program to display the course name, course
//fee and duration of course using
//StringTokenizer class
import java.util.StringTokenizer;
class Courses
{
    static String str = "Course name = .NET;" + "Course fee = Rs.
                8000;" + "Course duration = 3 months.";
    public static void main(String args[ ])
    {
        StringTokenizer strtknr = new StringTokenizer(str,. "=;");
        while(strtknr.hasMoreTokens( ))
        {
            String key = strtknr.nextToken( );
            String value = strtknr.nextToken( );
            System.out.println(key + "\t" + value);
        }
    }
}
```

Output

```
Course name      .NET;
Course fee       Rs.8000;
Course duration  3 months.
```

Program 3.12.2 Displaying Course Name, Fee and Duration using StringTokenizer Class

3.13 BITSET CLASS

A BitSet class creates an array that can store the bit values. It implements the Cloneable interface.

BitSet Constructors

(1) BitSet( ).

(2) BitSet(int s).

The first constructor creates a default bit set. The second constructor creates a bit set of the size specified by 's'. The size of an array can grow dynamically as needed. By default all bits are initialized to zero.

BitSet Methods

(1) **void and(BitSet bs) :** This method performs the AND operation on the invoking BitSet object and the object specified by 'bs'. The result of AND operation is stored in the invoking object.

(2) **void or(BitSet bs) :** This method performs the OR operation on the invoking BitSet object and the object specified by 'bs'. The result is stored in the invoking object.

(3) **void xor(BitSet bs) :** This method perfoms the XOR operation on the invoking BitSet object and the object specified by 'bs'. The result is stored in the invoking object.

(4) **void clear( ) and**

void clear(int index)

The first form of clear( ) method assigns all bit values to zero. The second form of clear( ) method assigns the bit values specified by index to zero.

(5) **boolean get(int index) and**

BitSet get(int start, int end)

The first form of get( ) method returns the bit value at the specified index. The second form returns the subset of BitSet specified by start and end index.

(6) **void set(int index) and**

void set(int index, boolean bit)

These methods sets the value of a bit at the specified index. The boolean value 'bit' specifies the value to be set, the bit can be either true or false.

(7) **void set(int start, int end) and Void set(int start, int end, boolean bit) :** These methods sets the subset of bits from start to end-1. The boolean value 'bit' specifies the value to be set. It can be either true or false.

(8) **String toString( ) :** This method returns the equivalent string representation of the BitSet object.

Example

```
import java.util.BitSet;
class Bset
{
    public static void main(String args[])
    {
        BitSet bs1 = new BitSet(12);
        BitSet bs2 = new BitSet(12);
        for(int i=1; i<=12; i++)
        {
            if((i%6) != 0)
                bs1.set(i);
            if((i%4) == 0)
                bs2.set(i);
        }
        System.out.println("BS1 = " + bs1);
        System.out.println("BS2 = " + bs2);
        bs2.and(bs1);
        System.out.println("ANDSet = " + bs2);
        bs2.or(bs1);
        System.out.println("ORSet = " + bs2);
        bs1.clear( );
        bs2.clear(2);
        bs2.clear(1);
        bs2.clear(9);
        bs2.xor(bs1);
        System.out.println("XORSet = " + bs2);
    }
}
```

Output

```
BS1 = {1, 2, 3, 4, 5, 7, 8, 9, 10, 11}
BS2 = {4, 8, 12}
ANDSet = {4, 8}
ORSet = {1, 2, 3, 4, 5, 7, 8, 9, 10, 11}
XORSet = {3, 4, 5, 7, 8, 10, 11}
```

3.14 DATE CLASS

The Date class contains the current date and time.

Date Constructors

(1) Date( )

(2) Date(long mseconds)

The first form of constructor creates a default Date object with the current date and time. Second constructor takes an argument that equals the number of milliseconds that have elapsed since midnight 1st January, 1970.

Date Methods : Some of the methods defined by Date class are moved into Calendar and DateFormat classes and remaining methods are depricated. The new methods added by Java2 to the Date class are as follows,

(1) **long getTime( ) :** This method is used to get the number of milliseconds that have elapsed since midnight 1st January, 1970.

(2) **void setTime(long time) :** This method is used to set the time.

(3) **String toString( ) :** This method returns the string equivalent of the invoking Date object.

(4) **Object clone( ) :** This method is used to create the duplicate of the invoking Date object.

(5) **boolean equals(Object date) :** This method is used to determine whether the two objects i.e., the invoking Date object and the object specified by date are equal. It returns true if they are equal, otherwise returns false.

(6) **boolean before(Date date) :** This method returns true, if the invoking Date object contains a date that comes before the date specified by 'date' object. Otherwise it return false.

(7) **boolean after(Date date) :** This method returns true, if the invoking Date object contains a date that comes after the date specified by 'date' object. Otherwise, it return false.

(8) **int compareTo(Date date) :** This method compares the invoking Date object with the specified 'date' object. The values returned by this method are as shown in Table 5.14.1.

Table 3.14.1 Return Values of compareTo Method

S.No.	Return Value	Meaning
(1)	0	If both objects are equal.
(2)	Positive value	If the invoking objects comes after 'date'.
(3)	Negative value	If the invoking object comes before 'date'.

Example

```
import java.util.Date;
class DateTime
{
    public static void main(String args[ ])
    {
        long time;
        Date d = new Date( );
        System.out.println("The current date and time is : " + d);
        time = d.getTime( );
        System.out.println("Number of milliseconds since 1/1/1970 : " + time);
    }
}
```

Output

The current date and time is : Wed Feb 14 10:49:35 GMT 2007

Number of millisecond since 1/1/1970 : 1171450175703

Comparison of Dates : Two date objects can be compared in three ways,

(1) Find the time elapsed since January 1, 1970 using getTime( ) for both the objects and then compare their results.

(2) Using the methods before( ), after( ) and equal( ).

(3) Using the compareTo( ), method. This method is defined by the comparable interface implemented by Date class.

File Name : DateDemo.java

```
import java.util.Date;
public class DateDemo
{
    public static void main(String args[ ])
    {
        Date d = new Date( );
        System.out.println(d);                    //Thu Jan 01 07:30:33 GMT+05:30 2004
                                                  //to extract hours
        System.out.println(d.getHours( ));        //prints 10 to extract minutes
        System.out.println(d.getMinutes( ));      //prints 11 to extract seconds
        System.out.println(d.getSeconds( ));      //prints 40 to extract month
        System.out.println(d.getMonth( ));        //prints 0 (for Jan) to extract date
        System.out.println(d.getDate( ));         //to extract day
        System.out.println(d.getDay( ));          //prints 5 (for Friday) to extract year
        System.out.println(d.getYear( ));         //prints 107 (add 1900, the epoch
                                                    year))
                                                  //to extract time
        System.out.println(d.getTime( ));
        //prints 1168576900281. Prints milliseconds from Jan 1, 1990(called epoch time)
        }
}
```

Output

```
Fri. Jan. 12 10:11:40 Ist 2007
10
11
40
0
12
5
107
1168576900281
```

3.15 CALENDAR CLASS

The Calendar is an abstract class. It provides a set of methods to manipulate the date and time. The subclasses of Calendar class provides the specific implementation to the abstract methods defined by Calendar to meet their own requirements. For example, GregorianCalendar is a subclass of Calendar class.

The Calendar class does not define its constructor. Therefore an object of an abstract calendar class can't be created. The Calendar class has several instance variables. These are declared as protected. They are as follows,

(1) **fieldsSet :** It is a boolean variable that indicates whether the time component is set or not.

(2) **fields :** It is a an array of integers that contains all the time components such as year, month, day, hour, time and second.

(3) **isSet :** It is a boolean array that indicates whether a specific time component is set or not.

(4) **time :** It is a long variable that contains the current time for this invoking object.

(5) **isTimeSet :** It is a boolean variable that indicates whether the current time is set or not.

The Calendar class also defines some integer constants used for setting or getting the components. These are as shown in Table 3.15.1.

Table 3.15.1 Calender int Constants

AM	MARCH	SUNDAY	DAY_OF_WEEK_IN_MONTH
PM	APRIL	MONDAY	WEEK_OF_MONTH
AM_PM	MAY	TUESDAY	WEEK_OF_YEAR
HOUR	JUNE	WEDNESDAY	MONTH
HOUR_OF_DAY	JULY	THURSDAY	YEAR
MINUTE	AUGUST	FRIDAY	ERA
SECOND	SEPTEMBER	SATURDAY	UNDECIMBER
MILLISECOND	OCTOBER	DAY_OF_WEEK	DST_OFFSET
JANUARY	NOVEMBER	DAY_OF_MONTH	FIELD_COUNT
FEBRUARY	DECEMBER	DAY_OF_YEAR	ZONE_OFFSET

Calendar Methods : The Calendar class defines several methods. Some commonly used methods are as follows,

(1) **static Calendar getInstance( ) :** This method returns the object of Calendar class for the default location and time zone.

(2) **int get(int const) :** This method returns the value of the requested component of the invoking object. This component is specified by 'const'. It can be any one of the integer constants.

Example : Calendar.DATE, Calendar.HOUR etc.

(3) **void set(int const, int value) :** This method sets the value of the date or time component specified by const. This 'const' must be one of the constant defined by Calender class.

Example : Calendar.HOUR.

(4) **TimeZone getTimeZone( ) and**

Void setTimeZone(TimeZone zObj)

These methods get and set the time zone for the invoking object. 'zObj' paramater of setTimeZone( ) method specifies the TimeZone object to be set.

(5) **boolean equals(Object calObj) :** This method returns true if both the invoking Calendar object and the one specified by calObj contains the same date. Otherwise it returns false.

(6) **boolean after(Object calObj) :** This method returns true if the invoking Calendar object contains a date that comes after the date specified by 'calObj' object. Otherwise it return false.

(7) **boolean before(Object calObj) :** This method returns true if the invoking Calendar object contains a date that comes before the date specified by 'calObj' object. Otherwise it returns false.

Example

```
import java.util.Calendar;
class DateAndTime
{
    public static void main(String args[ ])
    {
        Calendar cal = Calendar.getInstance( );
        System.out.print("Current date and time : ");
        System.out.println(cal.getTime( ));
        cal.set(2006, 11, 25, 3, 35);          //Set time to Dec. 25, 2006 to 3 : 35
```

```
        System.out.println("Updated time : " + cal.getTime( ));
        cal.set(Calendar.HOUR, 10);
        cal.set(Calendar.MINUTE, 40);
        cal.set(Calendar.SECOND, 05);
        System.out.print("Updated time : ");
        System.out.print(cal.get(Calendar.HOUR) + ":");
        System.out.print(cal.get(Calendar.MINUTE) + ":");
        System.out.println(cal.get(Calendar.SECOND));
        cal.set(cal.DATE, 20);
        System.out.println("Set date to 20 : " + cal.getTime( ));
    }
}
```

Output

```
Current date and time : Wed Feb 14 10:54:25 GMT 2007
Updated time : Thu Dec. 25 03:35:25 GMT 2007
Updated time : 10:40:5
Set date to 20 : Sat Jan 20 10:40:05 GMT 2007
```

3.16 TIMER CLASS

Timer and TimerTask : Both timer and TimerTask classes are defined in 'java.util' package. They are used for task scheduling. The scheduled task is then executed in the near future. In order to schedule the task, an instance of type TimerTask must be created. This instance can be then be scheduled for execution by a Timer object.

Constructors and Methods of TimerTask : TimerTask class implements Runnable interface. Thus, its constructor can be used to create a task (or a thread of execution). It has the following constructor.

TimerTask( )

TimerTask : Defines three methods,

(1) **boolean cancel( ) :** The method is used for termination of timertask. The cancel( ) method returns true if the timertask is not executed. Else, it returns false.

(2) ***abstract void run( ) :*** This is an abstract method defined by Runnable interface. The run( ) method contains the timertask execution code.

(3) **long scheduledExecutionTime( ) :** This method returns the most recent time of timertasks at which the task was scheduled for execution.

Constructors and methods of Timer : There are four constructors of Timer,

Timer( ) : This constructor is used for creation of an object of type Timer which executed as a normal thread.

timer(boolean isDmnThrd) : This constructor creates a new timer object that runs as a deamon thread if isDmnThrd is true.

Timer(String thName) : This constructor assigns a name to the instance of Timer.

Timer(String thName, boolean isDmnThrd) : This constructor assigns a name to the timer thread as well as sets the thread to be daemon thread if isDmnThrd is true.

The methods defined by Timer are,

(1) void cancel( )

This method terminates the Timer object.

(2) int purge( )

This method removes all the terminated Timer objects from the queue of the Timer.

(3) void schedule(TimerTask Timertask, long delay)

This method is used for scheduling timetask with an initial delay specified by delay.

(4) void schedule (TimerTask timertask, long delay, long period)

This method schedules the 'timertask' for repeated execution with an initial delay specified by delay. The timertask is then repeatedly executed at the specified period.

(5) void schedule(TimerTask timertask, Date time)

This method is used for scheduling the 'timertask' for execution at the specified time.

(6) void scheduler(TimerTask timertask, Date timer, long period)

This method schedules the timer task for execution at the specified time. After this, the timertask is then repeatedly executed at the specified period.

(7) void scheduleAtFixedRate (TimerTask timertask, long delay, long period)

This method schedules the timertask for execution with an initial delay specified by delay. The timertask is then repeatedly executed at the specified period and at a fixed rate.

(8) void scheduleAtFixedRate(TimerTask timertask, Date time, long period)

Here, scheduling of timertask is done for repeated fixed-rate execution at the specified time and after the specified period.

Example : Following program illustrates the usage of Timer and TimerTask classes.

File Name : TimerAndTimerTask.java

```
import java.util.*;
class MyTask extends TimerTask                    //creates a timer task
{
    int count = 1;
    public void run( )                            //overrides run( ) method
    {
        System.out.println("Congrats, Timer is called "+count+++"times");
    }
}
public class TimerAndTimerTask
{
    public static void main(String args[ ])
    {
        MyTask mtask = new MyTask( );
        Timer timer = new Timer( );
        //Schedule the task using timer instance as to repeat for every 2 secs with an
          initial delay 1 sec.
        timer.schedule(mtask, 1000, 2000);
        try
        {
            Thread.sleep(10000);
        }
        catch(InterruptedException e)
            {
                e.printStackTrace( );
        }
        timer.cancel( );                          //cancel the timer thread
    }
}
```

Output

```
congrats, timer is called 1 times
congrats, timer is called 2 times
congrats, timer is called 3 times
congrats, timer is called 4 times
congrats, timer is called 5 times
```

3.17 RANDOM CLASS

The random class generates pseudorandom numbers i.e., a sequence of random numbers generated by performing complex mathematical calculations. The random class uses the current time (which changes constantly) for generating random numbers, produce different sequence of random values each time the program is executed.

Random Constructors

(1) Random( ).

(2) Random(long seed).

The first form of constructor is a default number generator that uses the current time of day as the seed value. The second constructor takes the seed value as an argument that indicates the starting point for the random sequence. If the same seed value is used to initialize another random object. Then it generates the same random sequence. To generate different sequence the seed value should be different. The best way is to use the default constructor as it takes the current time, it generates the different sequence.

The Random class can generate random boolean, byte, float, double, int, long and Gaussian values.

Random Class Methods : The random class defines the following methods to generate nine types of random numbers,

(1) **boolean nextBoolean( ) :** This method returns the next random number of type boolean.

(2) **void nextBytes(byte value[ ]) :** This method generates the random numbers of type byte and store it in the byte array values.

(3) **double nextDouble( ) :** This method returns the next random number of type double.

(4) **float nextFloat( ) :** This method returns the next random number of type float.

(5) **double nextGaussian( ) :** This method returns the next random number of type Gaussian.

(6) **int nextInt( ) :** This method returns the next random number of type int.

(7) **int nextInt(int range) :** This method returns the next random number within the range zero to range of type int.

(8) **long nextLong( ) :** This method returns the next random number of type long.

(9) **void setSeed(long seed) :** This method sets the seed value to seed, as the starting point for the random number generator.

The below program generates random integers ranging from 0-4, i.e., 0, 1, 2, 3.

Example

```
import java.util.Random;
class IntRandomNums
{
    public static void main(String args[ ])
    {
        Random r = new Random( );
        int num;
        System.out.println("Randomly generated numbers are : ");
        for(int i=1; i<=16; i++)
        {
            num = 1+r.nextInt(4);       //generates random integers from 0 to 4
            System.out.print(" " +num);
            if(i%4==0)
            System.out.println( );
        }
    }
}
```

Output

```
Randomly generated numbers are :
1 2 2 1
4 2 3 4
4 2 4 1
4 2 1 1
```

❖ ❖ ❖

EXPECTED UNIVERSITY QUESTIONS WITH ANSWERS

Q1) List the important classes and interfaces of java.lang package.

Answer :

Refer Page No. 3.2, Section No. 3.1.

Q2) What are wrapper classes?

Answer :

Refer Page No. 3.3, Section No. 3.1.1.

Q3) How a double differ from double?

Answer :

Refer Page No. 3.7, Section No. 3.1.3.

Q4) Explain the usage of clone( ) and the cleanable interface with example.

Answer :

Refer Page No. 3.12, Section No. 3.1.7.

Q5) Write a short note on comparable interface.

Answer :

Refer Page No. 3.13, Section No. 3.1.8.

Q6) Briefly describe about collection interfaces?

Answer :

Refer Page No. 3.19, Section No. 3.3.

Q7) With an example explain the following collection classes.

(i) LinkedList.

(ii) HashSet.

(iii) TreeSet.

Answer :

Refer Page Nos. 3.24, 3.25 and 3.25, Section No. 3.4, Points 2, 3 and 5.

Q8) What is collection algorithm? Explain different collection algorithms available in collection framework.

Answer :

Refer Page No. 3.28, Section No. 3.5.

Q9) What is the purpose of interator class? How iterator class in used?

Answer :

Refer Page No. 3.32, Section No. 3.6.

Q10) Explain map interfaces with suitable examples.

Answer :

Refer Page No. 3.34, Section No. 3.8.

Q11) What are legacy classes? Explain methods of in each of those in detail.

Answer :

Refer Page No. 3.46, Section No. 3.11.

Q12) Write a short note on StringTokenier class with suitable example.

Answer :

Refer Page No. 3.58, Section No. 3.12.

Q13) Briefly describe about the following classes,

(i) BitSet.

(ii) Date.

(iii) Timer.

Answer :

Refer Page Nos. 3.61, 3.63 and 3.68, Section Nos. 3.13, 3.14 and 3.16.

❖ ❖ ❖