Packages

One of the biggest asset of JAVA is, a rich library set is available. These libraries include several pre-return methods of classes that can be used by everybody. These libraries include classes and interfaces for **IO Operations, Mathematical Operations, Networking, Graphics progamming** and many more activities. Depending on their functions, the classes and interfaces in these libraries are grouped together and called as packages.

11.1 The import statement

You can use java packages in a program by using **`import`** Keyword.

```
1. import <package name>.*;
2. import <package name>.class name/interface;
```

The * at the end of the import statement indicates that all the classes and interfaces of a given package are imported.
The second statement indicates that only the specified class from the given package as to be imported.

11.2 Package Types

Package	Purpose
java.lang	Default Core package
java.util	Data Structure programming like LinkedList, Stack, Vector, Date..
java.io	Basic IO operations and File handling
java.net	Network Programming.
java.applet	Create secured Web Pages
java.awt	Graphics,Labels, TextFields, Buttons, Checkboxs, Choices, Listboxs, Menus…
java.sql	Connect to database like Oracle,MS-SQL,MS-Access

11.2.1 Package `java.lang`

The java.lang package is one of the most important packages in Java. It provides a number of classes and interfaces that are fundamental to java programming.

`java.lang` Package includes following classes

Boolean	**Long**	**String**
Byte	**Math**	**StringBuffer**
Character	**Number**	**System**
Class	**Object**	**Thread**
ClassLoader	**Package**	**ThreadGroup**
Compiler	**Process**	**ThreadLocal**
Double	**Runtime**	**Throwable**
Float	**RuntimePermission**	**Void**
InheritableThreadLocal	**SecurityManager**	
Integer	**Short**	

Interfaces in `java.lang` Package

- **Cloneable**
- **Comparable**
- **Runnable**

11.3 Wrapper Classes

- In java there is an equalent class for each primary data type called as Wrapper classes. For Example : **`Integer`** class for **`int`** data type
- Wrapper classes are used to obtain bit-size, minimum, maximum data range, and also converts and manipulates **objects** to **basic datatypes** and vice versa.

11.3.1 The Byte class

The **`Byte`** class wraps a value of primitive type byte in an object. An object of type Byte contains a single field whose type is **`byte`**.

In addition, this class provides several methods for converting a **`byte`** to a **`String`** and a **`String`** to a **`byte`**, as well as other constants and methods useful when dealing with a **`byte`**.

Constructor Summary	
Byte(byte value)	Constructs a newly allocated Byte object that represents the specified byte value.
Byte(String s)	Constructs a newly allocated Byte object that represents the byte value indicated by the String parameter.

Constants	Purpose	Example
`static int MIN_VALUE`	returns minimum value of byte datatype	-128
`static int MAX_VALUE`	returns maximum value of byte datatype	+127
`static int SIZE`	returns size of byte datatype in bits	8
`static Class TYPE`	returns its data type	byte

Method Summary	
`byte`	**byteValue**() Returns the value of this Byte as a byte.
`int`	**compareTo**(Byte anotherByte) Compares two Byte objects numerically.
`double`	**doubleValue**() Returns the value of this Byte as a double.
`Boolean`	**equals**(Object obj) Compares this object to the specified object.
`float`	**floatValue**() Returns the value of this Byte as a float.
`int`	**intValue**() Returns the value of this Byte as an int.
`long`	**longValue**() Returns the value of this Byte as a long.

static byte	**parseByte**(String s) Parses the string argument as a signed decimal byte.
short	**shortValue**() Returns the value of this Byte as a short.
String	**toString**() Returns a String object representing this Byte's value.
static String	**toString**(byte b) Returns a new String object representing the specified byte.
static Byte	**valueOf**(byte b) Returns a Byte instance representing the specified byte value.
static Byte	**valueOf**(String s) Returns a Byte object holding the value given by the specified String.

11.3.2 The Integer class

The **Integer** class wraps a value of the primitive type **int** in an object. An object of type **Integer** contains a single field whose type is **int.**

In addition, this class provides several methods for converting an **int** to a **String** and a String to an **int**, as well as other constants and methods useful when dealing with an **int**.

Constants	Purpose	Example
static int SIZE	returns size of byte in bits	32
static int MIN_VALUE	returns minimum value of int datatype	-2^{31}
static int MAX_VALUE	returns maximum value of int datatype	$+2^{31}-1$

Method Summary	
byte	**byteValue()** Returns the value of this Integer as a byte.
int	**compareTo**(Integer anotherInteger) Compares two Integer objects numerically.
double	**doubleValue()** Returns the value of this Integer as a double.
boolean	**equals**(Object obj) Compares this object to the specified object.
float	**floatValue**() Returns the value of this Integer as a float.
int	**intValue**() Returns the value of this Integer as an int.
long	**longValue**() Returns the value of this Integer as a long.
static int	**parseInt**(String s) Parses the string argument as a signed decimal integer.

static int	**reverse**(int i) Returns the value obtained by reversing the order of the bits in the two's complement binary representation of the specified int value.
short	**shortValue**() Returns the value of this Integer as a short.
static String	**toBinaryString**(int i) Returns a string representation of the integer argument as an unsigned integer in base 2.
static String	**toHexString**(int i) Returns a string representation of the integer argument as an unsigned integer in base 16.
static String	**toOctalString**(int i) Returns a string representation of the integer argument as an unsigned integer in base 8.
String	**toString**() Returns a String object representing this Integer's value.
static String	**toString**(int i) Returns a String object representing the specified integer.
static Integer	**valueOf**(int i) Returns a Integer instance representing the specified int value.
static Integer	**valueOf**(String s) Returns an Integer object holding the value of the specified String.

Program

```
class IntegerClassDemo
{
      public static void main(String args[])
      {
      int num;
      num =  Integer.parseInt(args[0]);
      System.out.println("Binary = "+Integer.toBinaryString(num));
      System.out.println("Octal = "+Integer.toOctalString(num));
      System.out.println("Hexa Decimal = "+Integer.toHexString(num));
      }
}
```

OUTPUT

```
c:\> java IntegerClassDemo 15
Binary = 1111
Octal = 17
Hexa Decimal = f
```

11.3.3 The Long class

```
long l = Long.parseLong(String str)
```

Parses (converts) the string argument as a float value.

Note

The Long Class also having toBinaryString(), toHexString(), toOctalString() methods.

11.3.4 The Float class

```
float f=Float.parseFloat(String s);
```

Parses the string argument as a float value.

Program

```
//program to convert String object to float type
class FloatClassDemo
{
      public static void main(String args[])
      {
            float a,b,c;
            a=Float.parseFloat(args[0]);
            b=Float.parseFloat(args[1]);
            c=a+b;
            System.out.println("Sum of two number is "+c );
      }
}
```

Output

```
c:\>java FloatClassDemo 12.34 15.21
Sum of two numbers is 27.55
```

11.3.5 The Double class

```
double=Double.parseDouble(String s);
```

Parses the string argument as a double value.

11.4 Class Math

The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Class Hierarchy

```
java.lang.Object
  └java.lang.Math
```

```
public final class Math extends Object
```

- In **Math** class all the members (methods and constants) are declared as **static**. so **Math** class members can access using class name instead of its object name

Constants	Purpose	Example
static double PI	Returns pie value	3.14
static double E	The double value that is closer than any other to *e*, the base of the natural logarithms.	2.7182818

Method Summary	
static double	**abs**(double a) Returns the absolute value of a double value.

static float	**abs**(float a) Returns the absolute value of a float value.
static int	**abs**(int a) Returns the absolute value of an int value.
static long	**abs**(long a) Returns the absolute value of a long value.
static double	**acos**(double a) Returns the arc cosine of an angle, in the range of 0.0 through *pi*.
static double	**asin**(double a) Returns the arc sine of an angle, in the range of -*pi*/2 through *pi*/2.
static double	**atan**(double a) Returns the arc tangent of an angle, in the range of -*pi*/2 through *pi*/2.
static double	**atan2**(double y, double x) Converts rectangular coordinates (x, y) to polar (r, *theta*).
static double	**ceil**(double a) Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.
static double	**cos**(double a) Returns the trigonometric cosine of an angle.
static double	**cosh**(double x) Returns the hyperbolic cosine of a double value.
static double	**exp**(double a) Returns Euler's number *e* raised to the power of a double value.
static double	**floor**(double a) Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.
static double	**log**(double a) Returns the natural logarithm (base *e*) of a double value.
static double	**log10**(double a) Returns the base 10 logarithm of a double value.
static double	**max**(double a, double b) Returns the greater of two double values.
static float	**max**(float a, float b) Returns the greater of two float values.
static int	**max**(int a, int b) Returns the greater of two int values.
static long	**max**(long a, long b) Returns the greater of two long values.
static double	**min**(double a, double b) Returns the smaller of two double values.
static float	**min**(float a, float b) Returns the smaller of two float values.

static int	**min**(int a, int b) Returns the smaller of two int values.
static long	**min**(long a, long b) Returns the smaller of two long values.
static double	**pow**(double a, double b) Returns the value of the first argument raised to the power of the second argument.
static double	**random**() Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

static long	**round**(double a) Returns the closest long to the argument.
static int	**round**(float a) Returns the closest int to the argument.
static double	**sin**(double a) Returns the trigonometric sine of an angle.
static double	**sinh**(double x) Returns the hyperbolic sine of a double value.
static double	**sqrt**(double a) Returns the correctly rounded positive square root of a double value.
static double	**tan**(double a) Returns the trigonometric tangent of an angle.
static double	**tanh**(double x) Returns the hyperbolic tangent of a double value.
static double	**toDegrees**(double angrad) Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
static double	**toRadians**(double angdeg) Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

Program

```
class MathDemo
{
	public static void main(String args[])
	{
	double  x,sq;
	x=Double.parseDouble(args[0]);
	sq=Math.sqrt(Math.abs(a));
	System.out.println("Square root of "+x+" is "+sq);
	System.out.println("Floor = "+Math.floor(x));
	System.out.println("Ceil  = "+Math.ceil(x));
	System.out.println("Round = "+Math.round(x));
	double sinr=Math.sin(x);
	System.out.println("Sin in radians = "+sinr);
	double d=Math.toRadians(x);
	double sind=Math.sin(d);
```

```
        System.out.println("Sin in degrees = "+sind);
        }
}
```

Output

```
c:\>java MathDemo 30.453

Square root of 30.453 is 5.518423687974674
Floor = 30.0
Ceil  = 31.0
Round = 30
Sin in radians = -0.8208664847059639
Sin in degrees = 0.506831393787027
```

11.5 Character Class

The **Character** class wraps a value of the primitive type **char** in an object. An object of type **Character** contains a single field whose type is **char**.

In addition, this class provides several methods for determining a character's category (lowercase letter, digit, etc.) and for converting characters from uppercase to lowercase and vice versa.

Character information is based on the Unicode Standard.

Method Summary	
static boolean	**isDigit**(char ch) Determines if the specified character is a digit.
static boolean	**isLetter**(char ch) Determines if the specified character is a character.
static boolean	**isLetterOrDigit**(char ch) Determines if the specified character is a letter or digit.
static boolean	**isLowerCase**(char ch) Determines if the specified character is a lowercase char
static boolean	**isUpperCase**(char ch) Determines if the specified character is a uppercase char
static boolean	**isSpaceChar**(char ch) **Deprecated.** Replaced by isWhitespace(char).
static boolean	**isWhitespace**(char ch) Determines if the specified character is a Unicode space char
static char	**toLowerCase**(char ch) Converts into lowercase character.
static char	**toUpperCase**(char ch) Converts into uppercase character.

Program

```
class DigitDemo
{
      public static void main(String args[])
      {
            char ch='a';
            if(Character.isDigit(ch))
            System.out.println("Character is a number ");
            else
            System.out.println("Character is not a number ");
      }
}
```

Output `Character is not a number`

12. String Class

12.1 Introduction

A combination of characters is a string. Strings are instances of the class **String**. They are real objects. Strings are **constant** in Java program their values cannot be changed after they are created.

For example:

```
String str = "abc";
```

When a String literal is used in the program, java automatically creates instances of the String class. String are unusual in this respect.

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Constructor Summary
String() Initializes a newly created String object so that it represents an empty character sequence.
String(char[] value) Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.
String(String original) Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.
String(StringBuffer buffer) Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

Program

```
//program for declaration of strings

class StringDemo
{
	public static void main(String args[])
	{
		String str1=new String("Palar computers");
		//or
		String str2 ="palar computer centre";
		System.out.println("String1 is "+str1);
		System.out.println("String2 is "+str2);
	}
}
```

Methods in String Class

12.2 length()

Returns the length of a string.

```
{
	String  str = "palar computer centre";
```

```
                System.out.println(str.length());
        }
```

OUTPUT 21

12.3 indexOf(char ch)

Returns the index within this string of the first occurrence of the specified character. It returns –1 if the character is not found in the string. It is case sensitive.

```
{
        String str= "Palar Computer Centre";
        System.out.println (str.indexOf('o'));
}
```

OUTPUT 7

12.4 indexOf(String str)

Returns the index within this string of the first occurrence of the specified substring

```
        {
                String str="Palar Computer Centre";
                System.out.println(str.indexOf("put"));
        }
```

OUTPUT 9

12.5 charAt (int index_value)

Returns the character at the index

```
        {
                String str = "palar computer centre"
                System.out.println(str.charAt(3));
        }
```

OUTPUT a

12.6 equals(String str)

Compares the string with current string object.

```
{
        String str1="palar computer";
        String str2="Palar computer";
        System.out.println(str1.equals(str2));
}
```

OUTPUT false

12.7 equalsIgnoreCase(String str)

Comparation is very similar to equals method, but as the name suggest, it ignores the case of the String.

```
        {
                System.out.println(str1.equalsIgnoreCase(str2));
        }
```

OUTPUT true

12.8 startsWith(String startString)

Checks the current object starts with the same sequence of characters

```
{
        String str = "palar computer";
        System.out.println(str.startsWith("palar"));
}
```

OUTPUT

```
true
```

12.9 endsWith(String endString)

Returns **`true`** if the current object ends with given string.

12.10 substring(int i)

Returns new substring containing all the characters from the current object starting from index 'i' till the end.

```
{
        String str="palar computer";
        System.out.println(str.substring(7));
}
```

OUTPUT

```
omputer
```

12.11 substring(int i, int j)

Returns new substring starting from the index position 'i' till the index position j (Not including the upper in the j^{th} position)

```
{
        System.out.println(str.substring(7,10));
}
```

OUTPUT

```
omp
```

12.12 toLowerCase()

Converts all of the characters in this String to lower case

```
{
        String str="PALAR COMPUTER CENTRE";
        System.out.println(str.toLowerCase());
}
```

OUTPUT

```
  palar computer centre
```

12.13 toUpperCase( )

Converts all of the characters in this String to upper case

```
{
        System.out.println(str.toUpperCase());
}
```

12.14 replace(char oldChar, char newChar)

Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`

```
{
        System.out.println("Palar Computer Centre");
        System.out.println(str.replace('a','e'));
}
```

OUTPUT `peler computer centre`

Method Summary	
`char`	**charAt**(int index) Returns the char value at the specified index.
`int`	**compareTo**(String anotherString) Compares two strings
`int`	**compareToIgnoreCase**(String str) Compares two strings ignoring case differences.
`String`	**concat**(String str) Concatenates the specified string to the end of this string.
`static String`	**copyValueOf**(char[] data) Returns a String that represents the character sequence in the array specified.
`Boolean`	**endsWith**(String suffix) Tests if this string ends with the specified suffix.
`Boolean`	**equals**(Object anObject) Compares this string to the specified object.
`Boolean`	**equalsIgnoreCase**(String anotherString) Compares this String to another String, ignoring case considerations.
`byte[]`	**getBytes**() Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
`int`	**indexOf**(int ch) Returns the index within this string of the first occurrence of the specified character.
`int`	**indexOf**(String str) Returns the index within this string of the first occurrence of the specified substring.
`int`	**lastIndexOf**(int ch) Returns the index within this string of the last occurrence of the specified character.
`int`	**lastIndexOf**(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
`int`	**length**() Returns the length of this string.
`String`	**replace**(char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
`String[]`	**split**(String regex) Splits this string around matches of the given regular expression.
`Boolean`	**startsWith**(String prefix) Tests if this string starts with the specified prefix.

String	**substring**(int beginIndex) Returns a new string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
char[]	**toCharArray**() Converts this string to a new character array.
String	**toLowerCase**() Converts all of the characters in this String to lower case using the rules of the default locale.
String	**toString**() This object (which is already a string!) is itself returned.
String	**toUpperCase**() Converts all of the characters in this String to upper case using the rules of the default locale.
String	**trim**() Returns a copy of the string, with leading and trailing whitespace omitted.
static String	**valueOf**(boolean b) Returns the string representation of the boolean argument.
static String	**valueOf**(char c) Returns the string representation of the char argument.
static String	**valueOf**(char[] data) Returns the string representation of the char array argument.
static String	**valueOf**(double d) Returns the string representation of the double argument.
static String	**valueOf**(float f) Returns the string representation of the float argument.
static String	**valueOf**(int i) Returns the string representation of the int argument.
static String	**valueOf**(long l) Returns the string representation of the long argument.
static String	**valueOf**(Object obj) Returns the string representation of the Object argument.

Program

```
//program to demonstrate methods in string class
class StringDemo
{
public static void main(String args[])
{
      String str1=new String("Palar computer");
      //or
      String str2="palar computer";
      System.out.println("String1 is :"+str1);
      System.out.println("String2 is : "+str2);
      System.out.println("Length of the String1 is : "+str1.length( ));
      System.out.println ("Index of o is :"+str1.indexOf('o'));
      System.out.println("Index of put is : "+str1.indexOf("put"));
      System.out.println("Char at position 3 is : "+str1.charAt(3));
      System.out.println("str1 = str2 is : "+str1.equals(str2));
```

```
System.out.println("Ignore str1 = str2 is : "+str1.equalsIgnoreCase(str2));
System.out.println("Starts at palar is : "+str1.startsWith("Palar"));
System.out.println("Sub string of 7 is : "+str1.substring(7));
System.out.println("Sub string of 7 to 10 is : "+str1.substring(7,10));
System.out.println("Lower Case is : "+str1.toLowerCase());
System.out.println("Upper Case is : "+str1.toUpperCase());
System.out.println("Replace a to e is : "+str1.replace('a','e'));
System.out.println("Original String is : "+str1);
}
}
```

OUTPUT

```
String1 is : Palar computer
String2 is : palar computer
Length of the String1 is : 14
Index Of o is : 7
Index of put is : 9
Char at position 3 is : a
str1 = str2 is : false
Ignore str1 = str2 is : true
Starts at palar is : false
Sub string of 7 is : omputer
Sub string of 7 to  10 is : omp
Lower Case is : palar computer
Upper Case is : PALAR COMPUTER
Replace a to e is : Peler computer
Original String is :  Palar computer
```

12.2 StringBuffer Class

The original String object remains unchanged during all the string manipulations.

This is because, in Java, object of the String class, once created cannot be changed (immutable).

Java StringBuffer class, the changes will affect the original string (mutable).

Constructor Summary
StringBuffer() Constructs a string buffer with no characters in it and an initial capacity of 16 characters.
StringBuffer(int capacity) Constructs a string buffer with no characters in it and the specified initial capacity.
StringBuffer(String str) Constructs a string buffer initialized to the contents of the specified string.

Program

```
//program to declare StringBuffer

      class StringBufferDemo
      {
            public static  void main(String args[])
            {
            StringBuffer str=new StringBuffer("palar computer");
            System.out.println("String is "+str);
            }
      }
```

OUTPUT

```
palar computer
```

12.2.1 length()

This method returns the no. of characters in the buffer.

```
      {
            StringBuffer str=new StringBuffer("palar computer centre");
            System.out.println(str.length())
      }
```

OUTPUT

```
21
```

12.2.2 charAt(i)

It returns the character at the index i.

```
            System.out.println(str.charAt(2))
```

OUTPUT

```
1
```

12.2.3 append(char)

The method is used to add a character to the String buffer.

```
{
        StringBuffer str=new StringBuffer("palar computer");
        System.out.println(str.append('s'));
        System.out.println(str);
}
```

OUTPUT palar computers

12.2.4 append(String str)

This method is used to add a string.

```
{
        System.out.println(str.append("centre"));
        System.out.println(str);
}
```

OUTPUT palar computers centre

12.2.5 insert(i,ch)

This method is used to insert a character 'ch' at the index position 'i'.

```
{
        StringBuffer str;
        System.out.println(str.insert(7,'r'));
}
```

OUTPUT palar cromputers centre

12.2.6 insert (i,str)

This method insert a string str at the index position i.

12.2.7 reverse()

It reverse the given String Palar

```
System.out.println(str.reverse());
```

12.2.8 setCharAt(i,ch)

The method replaces the char at index position 'i' in the String buffer with the

```
{
        StringBuffer Str=new StringBuffer("palar computer");
        Str.setCharAt(0,'b');
        System.out.println(str);
```

```
}
```

OUTPUT

```
balar crmputer
```

Method Summary	
StringBuffer	**append**(boolean b) Appends the string representation of the boolean argument to the sequence.
StringBuffer	**append**(char c) Appends the string representation of the char argument to this sequence.
StringBuffer	**append**(char[] str) Appends the string representation of the char array argument to this sequence.
StringBuffer	**append**(double d) Appends the string representation of the double argument to this sequence.
StringBuffer	**append**(float f) Appends the string representation of the float argument to this sequence.
StringBuffer	**append**(int i) Appends the string representation of the int argument to this sequence.
StringBuffer	**append**(long lng) Appends the string representation of the long argument to this sequence.
StringBuffer	**append**(Object obj) Appends the string representation of the Object argument.
StringBuffer	**append**(String str) Appends the specified string to this character sequence.
StringBuffer	**append**(StringBuffer sb) Appends the specified StringBuffer to this sequence.
int	**capacity**() Returns the current capacity.
char	**charAt**(int index) Returns the char value in this sequence at the specified index.
StringBuffer	**delete**(int start, int end) Removes the characters in a substring of this sequence.
StringBuffer	**deleteCharAt**(int index) Removes the char at the specified position in this sequence.
int	**indexOf**(String str) Returns the index within this string of the first occurrence of the specified substring.
int	**indexOf**(String str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
StringBuffer	**insert**(int offset, char c) Inserts the string representation of the char argument into this sequence.

StringBuffer	**insert**(int offset, char[] str) Inserts the string representation of the char array argument into this sequence.
StringBuffer	**insert**(int offset, double d) Inserts the string representation of the double argument into this sequence.
StringBuffer	**insert**(int offset, Object obj) Inserts the string representation of the Object argument into this character sequence.
StringBuffer	**insert**(int offset, String str) Inserts the string into this character sequence.
int	**lastIndexOf**(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
int	**length**() Returns the length (character count).
StringBuffer	**replace**(int start, int end, String str) Replaces the characters in a substring of this sequence with characters in the specified String.
StringBuffer	**reverse**() Causes this character sequence to be replaced by the reverse of the sequence.
void	**setCharAt**(int index, char ch) The character at the specified index is set to ch.
String	**substring**(int start) Returns a new String that contains a subsequence of characters currently contained in this character sequence.
String	**substring**(int start, int end) Returns a new String that contains a subsequence of characters currently contained in this sequence.
String	**toString**() Returns a string representing the data in this sequence.

Program

```
//demonstration of StringBuffer class methods
class StringBufferDemo
{
public static  void main(String args[])
{
StringBuffer str=new StringBuffer("palar computer");
System.out.println(str);
```

```
            System.out.println("Length is -> "+str.length( ));
            System.out.println("Char at 2 is  -> "+str.charAt(2));
            System.out.println("Append s to string is ->
      "+str.append('s'));
            System.out.println("Append Centre is -> "+str.append("
      centre"));
            System.out.println("Insert r to 7th position is ->
                                                      "+str.insert(7,'r'));
            str.setCharAt(3,'z');
            System.out.println("Set z to 3th poisition is ->"+str);
            System.out.println("Reverse of string is ->"+str.reverse( ));
      }
}
```

OUTPUT

```
palar computer
Length is 14
Char at 2 is l
Append s to string is palar computers
Append Centre is palar computers centre
Insert r to 7th position is palar cromputers centre
Set z to 3th poisition is palzr cromputers centre
Reverse of string is ertnec sretupmorc rzlap
```

13. Java.util package

Collections Framework

A Collection is an object that groups multiple elements into a single unit. Collection are used to store, retrieve and manipulate data and to transmit data from one method to another.
A Collection framework is a unified architecture for representing and manipulating Collections.

Advantages of Collection Framework

- Reduces programming effort by providing useful data structure and algorithms.
- Increase program speed and quality since the implementation of each interface are interchangeable
- Allows interoperability among unrelated API
- Extending or adapting a collection easy
- The Collection interface is at the top of the hierarchy. It enables us to work with groups of objects.
- Encourage software reuse since the interface and algorithm are reusable

13.1 Collection Classes

The standard Collection Framework classes are

1. **Arrays**
2. **ArrayList**
3. **LinkedList**
4. **Date**
5. **HashSet**
6. **HastTable**
7. **TreeSet etc.,**

Algorithms are static methods within the Collection class. These method include methods for **sorting, searching, shuffling, data manipulations** etc.

Legacy Classes & Interface

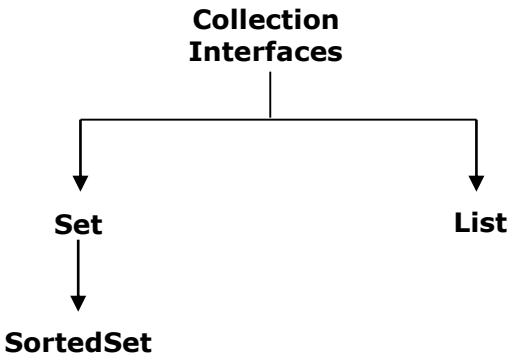
The legacy classes are **Dictionary, HaseTable, Properties, Stack and Vector**.

The legacy interface is the **Enumeration** Interface.

13.2 Util Interface

13.2.1 The List Interface

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



13.2.2 The Set Interface

The Set interface extends the Collection interface and declares the behavior of a collection that **dose not allow duplicate elements.** Therefore, the `add()` method returns false if an attempt is made to add duplicate elements to a set

13.2.3 The SortSet Interface

The extension of the **Set** interface is the **SortedSet Interface**. It declares the behavior of a set sorted in ascending order.

13.2.4 Exception

A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in set.

13.3 Arrays Class

Java 2 added a new class to **java.util.** called Arrays. This class provides various methods that are useful when working with arrays. Although these methods technically aren't part of the collection framework, they help bridge the gap between collections and arrays.

Method Summary	
`static int`	**binarySearch**(float[] a, float key) Searches the specified array of floats for the specified value using the binary search algorithm.
`static int`	**binarySearch**(int[] a, int key) Searches the specified array of ints for the specified value using the binary search algorithm.
`static boolean`	**equals**(float[] a, float[] a2) Returns true if the two specified arrays of floats are *equal* to one another.
`static boolean`	**equals**(int[] a, int[] a2) Returns true if the two specified arrays of ints are *equal* to one another.
`static boolean`	**equals**(Object[] a, Object[] a2) Returns true if the two specified arrays of Objects are *equal* to one another.
`static void`	**fill**(char[] a, char val) Assigns the specified char value to each element of the specified array of chars.
`static void`	**fill**(int[] a, int val) Assigns the specified int value to each element of the specified array of ints.
`static void`	**sort**(char[] a) Sorts the specified array of chars into ascending numerical order.
`static void`	**sort**(char[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of chars into ascending numerical order.

static void	**sort**(int[] a) Sorts the specified array of ints into ascending numerical order.
static void	**sort**(int[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of ints into ascending numerical order.
static String	**toString**(float[] a) Returns a string representation of the contents of the specified array.
static String	**toString**(int[] a) Returns a string representation of the contents of the spcified array.

Program

```
import java.util.*;
class ArraysDemo
{
      public static void main(String args[])
      {
            int a[]={90,25,10,30,15,60,95,100};
            Arrays.sort(a);
            for(int i:a)
            {
                  System.out.print(i+" ");
            }
                  //search an element in an array
                  int p=Arrays.binarySearch(a,15);
                  if(p>=0)
                  System.out.println("\nElement   15   found   at   position
            "+(++p));
                  else
                  System.out.println("\nElement 15 not found.... ");

                  int b[]={90,25,10,30,15,60,95,100};
                  Arrays.sort(b);
                  if(Arrays.equals(a,b))
                  System.out.println("Arrays are equal ");
                  else
                  System.out.println("Arrays are not equal ");
            }
      }
```

Output

```
10 15 25 30 60 90 95 100
Element 15 found at position 2
Array are equal
```

13.4 ArrayList Class

The **ArrayList** class extends **AbstractList** and implements the **List Interface.**

ArrayList supports dynamic arrays that can grow as needed.

In java, standard array are a fixed length is cannot be grown or shrink, that means static array, know in advance how may elements are array will hold.

In essence, an ArrayList is a variable-length array of object references. ie, ArrayList can dynamically increase or decrease in size. Array lists are created with an initial size. When the size is exceeded, the collection is automatically enlarged. When object are removed, the array may be shrunk.

Constructor Summary
ArrayList() Constructs an empty list with an initial capacity of ten.
ArrayList(Collection c) Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
ArrayList(int initialCapacity) Constructs an empty list with the specified initial capacity.

Method Summary	
boolean	**add**(E o) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specified Collection's Iterator.
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified Collection into this list, starting at the specified position.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this ArrayList instance.
boolean	**contains**(Object elem) Returns true if this list contains the specified element.
E	**get**(int inde x) Returns the element at the specified position in this list.
int	**indexOf**(Object elem) Searches for the first occurence of the given argument, testing for equality using the equals method.
boolean	**isEmpty**() Tests if this list has no elements.
int	**lastIndexOf**(Object elem) Returns the index of the last occurrence of the specified object in this list.
E	**remove**(int index) Removes the element at the specified position in this list.

boolean	**remove**(Object o) Removes a single instance of the specified element from this list, if it is present (optional operation).
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.
Object[]	**toArray**() Returns an array containing all of the elements in this list in the correct order.

Program

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

          // add elements to the array list
          al.add("C");
          al.add("A");
          al.add("E");
          al.add("B");
          al.add("D");
          al.add("F");
          al.add(1, "A2");

      System.out.println("Size of al after additions: " +al.size());

      // display the array list
      System.out.println("Contents of al: " + al);

      // Remove elements from the array list
      al.remove("F");
      al.remove(2);

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

Program **toArray()** is use to obtaining an Array from an ArrayList

// Convert an ArrayList into an array.

```
import java.util.*;
```

```
class ArrayListToArray
{
      public static void main(String args[])
      {
            // Create an array list
            ArrayList al = new ArrayList();

            // Add elements to the array list
            al.add(new Integer(1));
            al.add(new Integer(2));
            al.add(new Integer(3));
            al.add(new Integer(4));

            System.out.println("Contents of al: " + al);

            // get array
            Object ia[] = al.toArray();
            int sum = 0;

            // sum the array
            for(int i=0; i<ia.length; i++)
            sum += ((Integer) ia[i]).intValue();

            System.out.println("Sum is: " + sum);
      }
}
```

OUTPUT

```
Contents of al is : [1,2,3,4]
Sum is : 10
```

13.5 LinkedList Class

The **LinkedList** class extends **AbstractSquentialList** and implements the **List Interface**. It provides a Linked-list data structure.

Constructor Summary
LinkedList() Constructs an empty list.
LinkedList(Collection<? extends E> c) Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Method Summary	
boolean	**add**(E o) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
boolean	**addAll**(Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean	**addAll**(int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.

void	**addFirst**(E o) Inserts the given element at the beginning of this list.
void	**addLast**(E o) Appends the given element to the end of this list.
void	**clear**() Removes all of the elements from this list.
Object	**clone**() Returns a shallow copy of this LinkedList.
boolean	**contains**(Object o) Returns true if this list contains the specified element.
E	**element**() Retrieves, but does not remove, the head (first element) of this list.
E	**get**(int index) Returns the element at the specified position in this list.
E	**getFirst**() Returns the first element in this list.
E	**getLast**() Returns the last element in this list.
int	**indexOf**(Object o) Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
int	**lastIndexOf**(Object o) Returns the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
ListIterator<E>	**listIterator**(int index) Returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
E	**peek**() Retrieves, but does not remove, the head (first element) of this list.
E	**remove**() Retrieves and removes the head (first element) of this list.
E	**remove**(int index) Removes the element at the specified position in this list.
boolean	**remove**(Object o) Removes the first occurrence of the specified element in this list.
E	**removeFirst**() Removes and returns the first element from this list.
E	**removeLast**() Removes and returns the last element from this list.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.

Object[]	toArray() Returns an array containing all of the elements in this list in the correct order.

Program

```
import java.util.LinkedList;
class LinkedListDemo
{
	public static void main(String args[])
	{
		LinkedList l1=new LinkedList();
		l1.add("A");
		l1.add("B");
		l1.add("C");
		System.out.println(l1);
		l1.add(1,"D");
		System.out.println(l1);
		l1.addFirst("E");
		System.out.println(l1);
		l1.addLast("F");
		System.out.println(l1);
		l1.remove("B");
		system.out.println(l1);
		l1.removeFirst();
		System.out.println(l1);
		l1.removeLast();
		System.out.println(l1);
		System.out.println(l1.getFirst( ));
		System.out.println(l1.getLast( ));
		System.out.println(l1.size( ));
	}
}
```

OUTPUT

```
[A,B,C]
[A,D,B,C]
[E,A,D,B,C]
[E,A,D,B,C,F]
[E,A,D,C,F]
[A,D,E,F]
[A,D,C]
A
C
3
```

Program

// Modifing ListkedList elements using get() and set() methods.

```
import java.util.*;

class LinkedListDemo2
{
  public static void main(String args[])
 {
    // create a linked list
    LinkedList ll = new LinkedList();
```

```
    // add elements to the linked list
    ll.add("F");
    ll.add("B");
    ll.add("D");
    ll.add("E");
    ll.add("C");

    System.out.println("Original contents of ll: " + ll);

    // get and set a value
    Object val = ll.get(2);
    ll.set(2, (String) val + " Changed");

    System.out.println("ll after change: " + ll);
  }
}
```

13.6 The HashSet Class

HashSet class extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage. As most readers likely know, a hash table stores information by using a mechanism called **hashing**. In hashing, the information content of **key** is used to determine a unique value, called its **hash code.**

The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically, you never see the hash code itself. Also, your code can't directly index the hash table.

Constructor Summary
HashSet() Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).
HashSet(Collection<? extends E> c) Constructs a new set containing the elements in the specified collection.
HashSet(int initialCapacity) Constructs a new, empty set; the backing HashMap instance has the specified initial capacity.

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

Program

```
// Demonstrate HashSet.
import java.util.HashSet;

class HashSetDemo
{
  public static void main(String args[])
{
    // create a hash set
```

```
    HashSet hs = new HashSet();

    // add elements to the hash set
    hs.add("B");
    hs.add("A");
    hs.add("D");
    hs.add("E");
    hs.add("C");
    hs.add("F");
    if(hs.add("B"))
      System.out.println("B is added ");
    else
      System.out.println("Unable to add B, already exists..... ");

    System.out.println(hs);
    System.out.println(hs.contains("D"));
    hs.remove("A");
    System.out.println(hs);
    System.out.println(hs.size());

  }
}
```

OUTPUT

```
Unable to add B, already exists.....
[D, E, F, A, B, C]
true
[D, E, F, B, C]
5
```

13.7 TreeSet Class

TreeSet provides an implementation of the Set Interface that uses a tree for storage. Object are stored in sorted, ascending order. Access and retrieval times are quite fast. Which makes **TreeSet** an excellent choice when storing large amount of sorted information that must be found quickly.

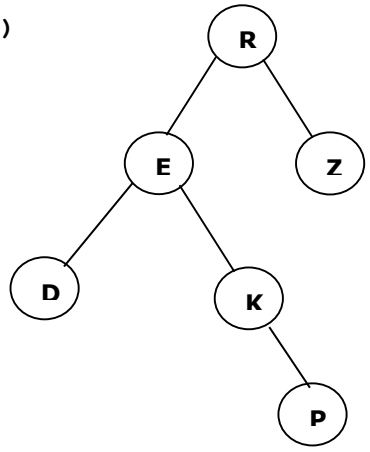
Constructor Summary
TreeSet() Constructs a new, empty set, sorted according to the elements' natural order.
TreeSet(Collection<? extends E> c) Constructs a new set containing the elements in the specified collection, sorted according to the elements' *natural order*.

Method Summary	
boolean	**add**(E o) Adds the specified element to this set if it is not already present.
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set.
void	**clear**() Removes all of the elements from this set.
Object	**clone**() Returns a shallow copy of this TreeSet instance.

boolean	**contains**(Object o) Returns true if this set contains the specified element.
E	**first**() Returns the first (lowest) element currently in this sorted set.
boolean	**isEmpty**() Returns true if this set contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
E	**last**() Returns the last (highest) element currently in this sorted set.
boolean	**remove**(Object o) Removes the specified element from this set if it is present.
int	**size**() Returns the number of elements in this set (its cardinality).
SortedSet<E>	**subSet**(E fromElement, E toElement) Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.

Program

```
import java.util.*;
class TreeSetDemo
{
      public static void main(String args[])
      {
            TreeSet t1=new TreeSet( );
            t1.add("R");
            t1.add("E");
            t1.add("D");
            t1.add("K");
            t1.add("P");
            t1.add("Z");
            System.out.println(t1);
            t1.remove("D");
            System.out.println(t1);
      }
}
```



OUTPUT

```
[D,E,K,P,R,Z]
[E,K,P,R,Z]
```

13.8 Iterator Interface

Iterator enables you to cycle through a collection, for example, you might want to display each element. By far, the easiest way to do this is to employ an Iterator, obtaining or removing elements.

Method Summary	
boolean	**hasNext**() Returns true if the iteration has more elements.
E	**next**() Returns the next element in the iteration.

void	**remove**() Removes from the underlying collection the last element returned by the iterator (optional operation).

13.9 ListIterator Interface

ListIterator extends **Iterator** to allow bi-directional traversal of a list, and the modification of elements.

Method Summary	
void	**add**(E o) Inserts the specified element into the list (optional operation).
boolean	**hasNext**() Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	**hasPrevious**() Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E	**next**() Returns the next element in the list.
int	**nextIndex**() Returns the index of the element that would be returned by a subsequent call to next.
E	**previous**() Returns the previous element in the list.
int	**previousIndex**() Returns the index of the element that would be returned by a subsequent call to previous.
void	**remove**() Removes from the list the last element that was returned by next or previous (optional operation).
void	**set**(E o) Replaces the last element returned by next or previous with the specified element .

Program

// Demonstrate iterators.

```
import java.util.*;
class IteratorDemo
{
  public static void main(String args[])
{
    // create an array list
    ArrayList al = new ArrayList();

    // add elements to the array list
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");

    // use iterator to display contents of al
    System.out.print("Original contents of al: ");
```

```
    Iterator itr = al.iterator();
    while(itr.hasNext())
{
      Object element = itr.next();
      System.out.print(element + " ");
}
    System.out.println();

    // modify objects being iterated
    ListIterator litr = al.listIterator();
    while(litr.hasNext())
{
      Object element = litr.next();
      litr.set(element + "+");
}

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

    // now, display the list backwards
    System.out.print("Modified list backwards: ");
    while(litr.hasPrevious())
 {
      Object element = litr.previous();
      System.out.print(element + " ");
 }
    System.out.println();
  }
}
```

13.10 The Legacy Classes and Interfaces

The original version of java.util did not included the collection framework. In general, the legacy classes are supported because a large base of code exists that uses them, including code still used by the Java 2 API

The Legacy Classes

- **Dictionary**
- **HashTable**
- **Properties**
- **Stack**
- **Vector**

The Legacy Interface

The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code.

Methods	Description
`boolean hasMoreElements()`	Returns true if there are more elements, otherwise returns false
`Object nextElement()`	Returns the next object in the enumeration as generic object reference

13.5.1 Vector Class

Vector implements a dynamic array, it is similar to ArrayList, but with two differences. Vector is synchronized, and it contains many legacy methods that are not part of the collections framework.

Constructor Summary
Vector() Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
Vector(int initialCapacity) Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
Vector(int initialCapacity, int capacityIncrement) Constructs an empty vector with the specified initial capacity and capacity increment.
Vector(Collection c) Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Method Summary	
`boolean`	**add**(obj) Appends the specified element to the end of this Vector.

void	**add**(int index, E element) Inserts the specified element at the specified position in this Vector.
void	**addElement**(obj) Adds the specified component to the end of this vector, increasing its size by one.
int	**capacity**() Returns the current capacity of this vector.
void	**clear**() Removes all of the elements from this Vector.
Object	**clone**() Returns a clone of this vector.
boolean	**contains**(Object elem) Tests if the specified object is a component in this vector.
Object	**elementAt**(int index) Returns the component at the specified index.
Enumeration<E>	**elements**() Returns an enumeration of the components of this vector.
boolean	**equals**(Object o) Compares the specified Object with this Vector for equality.
Object	**firstElement**() Returns the first component (the item at index 0) of this vector.
Object	**get**(int index) Returns the element at the specified position in this Vector.
int	**indexOf**(Object elem) Searches for the first occurence of the given argument, testing for equality using the equals method.
void	**insertElementAt**(Object obj, int index) Inserts the specified object as a component in this vector at the specified index.
boolean	**isEmpty**() Tests if this vector has no components.
Object	**lastElement**() Returns the last component of the vector.
int	**lastIndexOf**(Object elem) Returns the index of the last occurrence of the specified object in this vector.
Object	**remove**(int index) Removes the element at the specified position in this Vector.
boolean	**remove**(Object o) Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
void	**removeAllElements**() Removes all components from this vector and sets its size to zero.

boolean	**removeElement**(Object obj) Removes the first (lowest-indexed) occurrence of the argument from this vector.
void	**removeElementAt**(int index) Deletes the component at the specified index.
protected void	**removeRange**(int fromIndex, int toIndex) Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
Object	**set**(int index, E element) Replaces the element at the specified position in this Vector with the specified element.
void	**setElementAt**(E obj, int index) Sets the component at the specified index of this vector to be the specified object.
void	**setSize**(int newSize) Sets the size of this vector.
int	**size**() Returns the number of components in this vector.
Object[]	**toArray**() Returns an array containing all of the elements in this Vector in the correct order.
String	**toString**() Returns a string representation of this Vector, containing the String representation of each element.

Program

```
// Demonstrate various Vector operations.
import java.util.*;

class VectorDemo
{
  public static void main(String args[])
{


    Vector v = new Vector(3, 2);

    System.out.println("Initial size: " + v.size());
    System.out.println("Initial capacity: " +v.capacity());

    v.addElement(new Integer(1));
    v.addElement(new Integer(2));
    v.addElement(new Integer(3));
    v.addElement(new Integer(4));
    System.out.println("Vector elements are " + v);
    System.out.println("Capacity after four additions: " + v.capacity());
    System.out.println("Initial size: " + v.size());

   v.addElement(new Double(5.45));
         v.addElement(new Double(6.08));
         v.addElement(new Integer(7));

  System.out.println("Vector elements are " + v);
```

```
        System.out.println("Capacity    after    Sevent    additions:    "    +
    v.capacity());
        System.out.println("After  size: " + v.size());
           System.out.println("First          element:          "          +
    (Integer)v.firstElement());
        System.out.println("Last element: " +   (Integer)v.lastElement());

        if(v.contains(new Integer(3)))
          System.out.println("Vector contains 3.");

        // enumerate the elements in the vector one by one .
        Enumeration vEnum = v.elements();

        System.out.println("\nElements in vector:");

    while(vEnum.hasMoreElements())
         System.out.print(vEnum.nextElement() + " ");
      }
    }
```

13.5.2 HashTable Class

It is used to create hash table. The hash table contains two columns. The 1st column is the index and the 2nd is value.

Constructor Summary
Hashtable() Constructs a new, empty hashtable with a default initial capacity (11)
Hashtable(int initialCapacity) Constructs a new, empty hashtable with the specified initial capacity

Method Summary	
void	**clear**() Clears this hashtable so that it contains no keys.
Object	**clone**() Creates a shallow copy of this hashtable.
boolean	**contains**(Object value) Tests if some key maps into the specified value in this hashtable.
Enumeration<V>	**elements**() Returns an enumeration of the values in this hashtable.
boolean	**equals**(Object o) Compares the specified Object with this Map for equality, as per the definition in the Map interface.
V	**get**(Object key) Returns the value to which the specified key is mapped in this hashtable.
boolean	**isEmpty**() Tests if this hashtable maps no keys to values.
V	**put**(K key, V value) Maps the specified key to the specified value in this hashtable.

V	**remove**(Object key) Removes the key (and its corresponding value) from this hashtable.
int	**size**() Returns the number of keys in this hashtable.
String	**toString**() Returns a string representation of this Hashtable object in the form of a set of entries, enclosed in braces and separated by the ASCII characters ", " (comma and space).

Program

```
import java.util.*;
class HashTableDemo
{
	public static void main (String args[])
	{
		Hashtable h=new Hashtable( );
		h.put("100","kumar");
		h.put("101","suresh");
		h.put("102","Ram");
		h.put("103","Sri");
		System.out.println(h.get("102"));
		System.out.println(h);
	}
}
```

OUTPUT

```
Ram
{103  Sri   102 Ram     101=suresh        100=kumar}
```

13.5.3 Stack class

Stack is a subclass of **Vector** that implements a standard **last-in, first-out stack.** Stack only defines the constructor, which create an empty stack. Stack includes all the methods defined by Vector, and adds several of its own given below**.**

Constructor Summary
Stack() Creates an empty Stack.

Method Summary	
boolean	**empty**() Tests if this stack is empty.
E	**peek**() Looks at the object at the top of this stack without removing it from the stack.
E	**pop**() Removes the object at the top of this stack and returns that object as the value of this function.
E	**push**(E item) Pushes an item onto the top of this stack.
int	**search**(Object o) Returns the 1-based position where an object is on this stack.

Program

```
// Demonstrate the Stack class.
import java.util.*;

class StackDemo
{
  static void showpush(Stack st, int a)
{
    st.push(new Integer(a));
    System.out.println("push(" + a + ")");
    System.out.println("stack: " + st);
  }

  static void showpop(Stack st)
{
    System.out.print("pop -> ");
    Integer a = (Integer) st.pop();
    System.out.println(a);
    System.out.println("stack: " + st);
  }

  public static void main(String args[])
{
    Stack st = new Stack();

    System.out.println("stack: " + st);
    System.out.println("stack: " + st.size());
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpush(st, 19);
    showpush(st, 45);
int p=st.search(19);
if(p>=0)
System.out.println("Element 19 in stack at position "+p);
else
   System.out.println("Element 19 not in the stack ");
try
{
 showpop(st);
    showpop(st);
    showpop(st);
          showpop(st);
          showpop(st);
          showpop(st);
          }
catch (EmptyStackException e)
   {
          System.out.println("empty stack");
          }
  }
}
```

13.6 StringTokenizer Class

StringTokenizer class provides the first step in this parsing process, prasing is the division of text into a set of discrete part, or tokens, which in a certain sequence to convey a semantic meaning.

StringTokenizer implements the **Enumeration** interface.

Constructor Summary
StringTokenizer(String str) Constructs a string tokenizer for the specified string.
StringTokenizer(String str, String delim) Constructs a string tokenizer for the specified string.

Method Summary	
int	**countTokens**() Calculates the number of tokens
boolean	**hasMoreTokens**() Tests if there are more tokens available from this tokenizer's string.

String	**nextToken**() Returns the next token from this string tokenizer.
String	**nextToken**(String delim) Returns the next token in this string tokenizer's string

Program

```
// Demonstrate StringTokenizer.
import java.util.StringTokenizer;

class STDemo
{
static String email="palar@yahoo.com;dkm@gmail.com;gtech@rediff.com";

public static void main(String args[])
{
      StringTokenizer st = new StringTokenizer(email, "@;");
      System.out.println("Number of Tokens is "+st.countTokens());
      while(st.hasMoreTokens())
      {
      String id = st.nextToken();
      String sever= st.nextToken();
      System.out.println(id+ "\t" + sever);
      }
}
}
```

13.7 Date Class

The **Date** class encapsulates the current date and time. Date class is original version of Java1.0. When Java1.1 was released, many of the functions carried out by the original Date class were move into the **Calendar** and **DateFormat** classes, so many of the Date methods are deprecated (old).

Constructor Summary

Date() Allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.
Date(int year, int month, int date) **Deprecated.** *As of JDK version 1.1, replaced by Calendar.set(year + 1900, month, date) or GregorianCalendar(year + 1900, month, date).*
Date(long date) Allocates a Date object and initializes it to represent the specified number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT.
Date(String s) **Deprecated.** *As of JDK version 1.1, replaced by DateFormat.parse(String s).*

Method Summary	
boolean	**after**(Date when) Tests if this date is after the specified date.
boolean	**before**(Date when) Tests if this date is before the specified date.
Object	**clone**() Return a copy of this object.
int	**compareTo**(Date anotherDate) Compares two Dates for ordering.
boolean	**equals**(Object obj) Compares two dates for equality.
int	**getDate**() **Deprecated.** *As of JDK version 1.1, replaced by Calendar.get(Calendar.DAY_OF_MONTH).*
int	**getDay**() **Deprecated.** *As of JDK version 1.1, replaced by Calendar.get(Calendar.DAY_OF_WEEK).*
int	**getHours**() **Deprecated.** *As of JDK version 1.1, replaced by Calendar.get(Calendar.HOUR_OF_DAY).*
int	**getMinutes**() **Deprecated.** *As of JDK version 1.1, replaced by Calendar.get(Calendar.MINUTE).*
int	**getMonth**() **Deprecated.** *As of JDK version 1.1, replaced by Calendar.get(Calendar.MONTH).*
int	**getSeconds**() **Deprecated.** *As of JDK version 1.1, replaced by Calendar.get(Calendar.SECOND).*
long	**getTime**() Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.
int	**getYear**() **Deprecated.** *As of JDK version 1.1, replaced by Calendar.get(Calendar.YEAR) - 1900.*

static long	**parse**(String s) **Deprecated.** *As of JDK version 1.1, replaced by DateFormat.parse(String s).*
void	**setDate**(int date) **Deprecated.** *As of JDK version 1.1, replaced by Calendar.set(Calendar.DAY_OF_MONTH, int date).*
void	**setHours**(int hours) **Deprecated.** *As of JDK version 1.1, replaced by Calendar.set(Calendar.HOUR_OF_DAY, int hours).*
void	**setMinutes**(int minutes) **Deprecated.** *As of JDK version 1.1, replaced by Calendar.set(Calendar.MINUTE, int minutes).*
void	**setMonth**(int month) **Deprecated.** *As of JDK version 1.1, replaced by Calendar.set(Calendar.MONTH, int month).*
void	**setSeconds**(int seconds) **Deprecated.** *As of JDK version 1.1, replaced by Calendar.set(Calendar.SECOND, int seconds).*

void	**setTime**(long time) Sets this Date object to represent a point in time that is time milliseconds after January 1, 1970 00:00:00 GMT.
void	**setYear**(int year) **Deprecated.** *As of JDK version 1.1, replaced by Calendar.set(Calendar.YEAR, year + 1900).*
String	**toString**() Converts this Date object to a String of the form:

Program

```
// Show date and time using only Date methods.
import java.util.Date;
import java.text.*;

class DateDemo
{
  public static void main(String args[])
 {
    // Instantiate a Date object
    Date date = new Date();

    // display time and date using toString()
    System.out.println("System date = : "+date);

    SimpleDateFormat sd = new SimpleDateFormat("dd/MMMM/yyyy");
    String s1= sd.format(date);
    System.out.println("Formatted Date : "+s1);

    SimpleDateFormat st = new SimpleDateFormat("hh:mm:ss");
    String s2= st.format(date);
    System.out.println("Formatted Time : "+s2);

    Date dob= new Date(2007,06,21);
      Date doj = new Date(2009,06,21);

      if(doj.after(dob))
      System.out.println("Entry dates are correct ");
      else
      System.out.println("Pls. Enter Doj date after Dob date ");


  }
}
```

14. IO Package

The java.io.package provides Classes and Interfaces that supports for Input and Output Operations.

14.1 File class

It is used to obtain or manipulate the information associated with a disc file, such as a permission, directory path and so on.

Constructor Summary
File(File parent, String child) Creates a new File instance from a parent abstract pathname and a child pathname string.
File(String pathname) Creates a new File instance by converting the given pathname string into an abstract pathname.
File(String parent, String child) Creates a new File instance from a parent pathname string and a child pathname string.
File(URI uri) Creates a new File instance by converting the given file: URI into an abstract pathname.

Method Summary	
boolean	**canRead**() Tests whether the application can read the file denoted by this abstract pathname.
boolean	**canWrite**() Tests whether the application can modify the file denoted by this abstract pathname.
boolean	**createNewFile**() Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.
boolean	**delete**() Deletes the file or directory denoted by this abstract pathname.
boolean	**equals**(Object obj) Tests this abstract pathname for equality with the given object.
boolean	**exists**() Tests whether the file or directory denoted by this abstract pathname exists.
String	**getAbsolutePath**() Returns the absolute pathname string of this abstract pathname.
String	**getName**() Returns the name of the file or directory denoted by this abstract pathname.

String	**getParent**() Returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
File	**getParentFile**() Returns the abstract pathname of this abstract pathname's parent, or null if this pathname does not name a parent directory.
String	**getPath**() Converts this abstract pathname into a pathname string.
boolean	**isAbsolute**() Tests whether this abstract pathname is absolute.
boolean	**isDirectory**() Tests whether the file denoted by this abstract pathname is a directory.
boolean	**isFile**() Tests whether the file denoted by this abstract pathname is a normal file.
boolean	**isHidden**() Tests whether the file named by this abstract pathname is a hidden file.
long	**lastModified**() Returns the time that the file denoted by this abstract pathname was last modified.
long	**length**() Returns the length of the file denoted by this abstract pathname.
String[]	**list**() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
static File[]	**listRoots**() List the available filesystem roots.
boolean	**mkdir**() Creates the directory named by this abstract pathname.
boolean	**mkdirs**() Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories.
boolean	**renameTo**(File dest) Renames the file denoted by this abstract pathname.
boolean	**setReadOnly**() Marks the file or directory named by this abstract pathname so that only read operations are allowed.
String	**toString**() Returns the pathname string of this abstract pathname.

Syntax

```
File obj=new File("directory path/file name");
```

Program **To obtain File Properties**

```
import java.io.*;
class FileDemo
{
	public static void P(String s)
	{
	System.out.println(s);
	}
	public static void main(String arg[])
	{
		File f1= new File("f:/java/io/FileDemo.java");
		P("File name : " + f1.getName());
		P("Path : "+f1.getPath());
		P("Abs Path : "+f1.getAbsolutePath());
		P("Parent  : "+f1.getParent());
		P(f1.exists() ? "Exists " : "does not Exist ");
		P(f1.canWrite () ? "is writeable " : "is not writeable");
		P(f1.canRead () ? "is readable " :" is not readable");
		P("is " + (f1.isDirectory() ? "directory ": "Not  a directory
		"));
		P("is " + (f1.isHidden() ? "Hidden ": "Not  Hidden"));
		P(f1.isFile() ? "Normal file " : "might be a named pipe");
		P(f1.isAbsolute () ? "is obsolute " : " is not absolute ");
		P("File last modified : "+f1.lastModified());
		P("file size : "+f1.length() + "Bytes");
	}
}
```

Program To Rename a File

```
import java.io.*;
class FileRename
{
	public static void main(String args[])
	{
		File f1=new File("f:/java/first.java");
		File f2=new File("f:/java/second.java");
		f1.renameTo(f2);
	}
}
```

Program To Create a Directory

```
import  java.io.*;
class MakeDirectory
{
	public static void main(String args[])
	{
		File f1=new File("f:/java");
		if(!f1.mkdir())
		System.out.println("Folder already Exists..... ");
	}
}
```

Program To Delete a File

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

```
        {
                File f1=new File("f:/java");
                if(!f1.delete())
                System.out.println("File/Folder not found!.... ");

        }
```

Program To List the Files in a Directory

```
import java.io.*;
class FileList
{
        public static void main(String args[])
        {
                int i;
                File f=new File("f:/java/util");
                String str[]=f.list();
                for(String i:str)
                {
                        System.out.println(i);
                }
        }
}
```

14.3 Streams

Java program perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the java I/O systems.

All streams behave in the same manner, even if the actual physical devices to which they are linked differ.

Ie., an input stream can get from many different kinds of input devices like: a keyboard ,hard disk, or a network socket and, also for output stream.

14.3.1 Types of Streams in Java

- **Byte Streams** provide a convenient means for handling input and output of bytes.
- **Character Streams** provide a convenient means for handling input and output of Characters.
  They use Unicode and, therefore can be internationalized.

Note: Character streams were added in java1.1.
One other point: at the lowest level, all I/O is still byte-Oriented.

14.4 Input & Output Streams

Java's stream-based I/O is built upon four abstract Classes:

The Byte Steam Classes

- **InputStream**
- **OutputSream**

The two most important method are **`read()`** and **`write()`**, which read and write bytes of data.

The Character Stream Classes

- **Reader**
- **Writer**

The two important methods are **`read()`** and **`write()`**, which read and write Characters of data.

14.4.1 The Byte Stream Clasess

InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
BufferedInputStream	Buffered input Stream
BufferedOutStream	Buffered output Stream
DataInputStream	An input stream that contains methods for reading the java standard data types
DataOutputStream	An output stream that contains methods for writing the java standard data types
FileInputStream	Input Stream that reads from a file
FileOutputStream	Output Stream that writes to a file

14.4.2 The Character Stream classes

Reader	Abstract class that describes character stream input
Writer	Abstract class that describes character stream output
BufferedReader	Buffered input character Stream
BufferedWriter	Buffered output character Stream
FileReader	Input Stream that reads from a file
FileWriter	Output Stream that reads to a file
InputStreamReader	Input stream that translates bytes to characters
OutputStreamReader	Output stream that translates bytes to characters
LineNumberReader	Input Stream that counts lines
PrintWriter	Output Stream that contains print() and println()
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string

DataInputStream Class

This class help to create stream from keyboard to get input.

Program

```
import java.io.*;
class DataInputDemo
{
      public static void main(String args[])throws IOException
      {
            DataInputStream ds=new DataInputStream(System.in);
            String str;
            int n1,n2,n3;
            System.out.print("Enter the first no:");
            str=ds.readLine( );
```

```
                n1=Integer.parseInt(str);
                System.out.print("Enter the second no:");
                str=ds.readLine( );
                n2=Integer.parseInt(str);
                n3=n1+n2;
                System.out.println("Sum of two nos is "+n3);
          }
    }
```

OUTPUT

```
Enter the first no  : 100
Enter the Second no : 200
Sum of two nos is 300
```

Program for File Creation

```
import java.io.*;
class FileWrite
{
      public static void main(String args[]) throws IOException
      {
            InputStreamReader kb=new InputStreamReader(System.in);

            BufferedReader br=new BufferedReader (kb);
            FileOutputStream fw=new FileOutputStream(args[0]);
            char s;
      while((s=(char)br.read()) !='*')
      {
            fw.write((int) s);
      }
      fw.close();
}
}
```

Program To View the Content of the File

```
import java.io.*;
import java.util.*;
class FileRead
{
public static void main(String arg[]) throws IOException
{
Scanner s1=new Scanner(System.in);
System.out.println("Enter a file name to read ");
String file =s1.next();
try
{
FileInputStream fr = new FileInputStream(file);


int s;
while((s=fr.read())!=-1)
{
System.out.print((char)s);
}
fr.close();
}
catch(FileNotFoundException e)
{
      System.out.println("File not found ... ");
}
}

}
```

Program To Copying File From one to another

```
import java.io.*;
class FileCopy {
public static void main(String args[]) throws IOException
{
    int i;
    FileInputStream fin;
    FileOutputStream fout;

    try {
      // open input file
      try {
        fin = new FileInputStream(args[0]);
      } catch(FileNotFoundException e) {
        System.out.println("Input File Not Found");
        return;
      }

      // open output file
      try {
        fout = new FileOutputStream(args[1]);
      } catch(FileNotFoundException e) {
        System.out.println("Error Opening Output File");
        return;
      }
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Usage: CopyFile From To");
      return;
    }

    // Copy File
    try {
            while((i=fin.read())!=-1)
            fout.write(i);

    } catch(IOException e) {
      System.out.println("File Error");
    }

    fin.close();
    fout.close();
  }
                                        }
```

15. User Defined Packages

15.1 Packages

Package allows us to combine groups of classes and interfaces and also it controls the accessibility of class inside a specific package.

Package provide protection to classes, variables and methods in larger way than on a class-by-class basis.

Package can be used to identify classes and interfaces

Program

```
//package<folder_name>  example  c:\palar

package palar;
public class Arithmetic
{
      public static int sum(int a,int b)
      {
            return(a+b);
      }
      public static int diff(int a,int b)
      {
            return(a-b);
      }
      public static int mul(int a,int b)
      {
            return(a*b);
      }
      public static int div(int a,int b)
      {
            return (a/b);
      }
}
```

Creating Sub Package

```
//c:\palar\sub
package palar.sub;
public class MyMath
{
public long fact(int n)
{
long p=1;
for(int i=1;i<=n;i++)
p=p*i;
return p;
}
}
```

15.2 Accessing User defined package (palar)

```
import palar.Arithmetic;
import palar.sub.MyMath;
```

```
class AccessPalar
{
	public static void main(String args[])
	{
		MyMath m= new MyMath();
		int p=Arithmetic.mul(20,10);
		System.out.println("Product of two nos = "+p);
		System.out.println(m.fact(5));
	}
}
```

OUTPUT

```
		200
		120
```

15. 3 Scope of Access Specifiers

Class Member Access				
	Private	**No modifier**	**Protected**	**Public**
Same class	Yes	Yes	Yes	Yes
Same package sub class	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package Non-subclass	No	No	No	Yes

16. Thread

16.1 Introduction

Unlike most other computer languages, java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that **can run concurrently**

Each part of such a program is called a **Thread**, the each thread defines a separate path of execution. Thus, multithreading is specialized form of multitasking.

However, there are two distinct types of multitasking:

1. **Process-based**
2. **Thread-based**.

1. **A process-based** multitasking is the feature that allow your computer to turn two or more programs concurrently.

   For example: process-based multitasking enables you to run the java compiler at the same time that you are using a text editor(notepad)

2. A **thread-based** multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously.

   For example: a text editor can format text at the same time that it is printing, these two actions are performed by two separate thread.

Processes are **heavyweight** task, where as thread are **lightweight** task

A thread is similar to a sequential program. Like sequential program,a thread also has a beginning , and an end. However a thread is not a program on its own. But runs within a program. Every program has at least one thread that is called primary thread. You can create more thread when necessary.

These are two types of threaded applications.

1. **Single threaded applications**
2. **Multi threaded applications.**

16.2 Single Threaded Application

A process ie made up of only one thread is said to be single threaded. A single threaded application can perform only one task at a time. You have to wait for one task to be implemented before another can start.

16.3 Multi Threaded Applications

A process having more than one thread is said to be multi threaded. A multi threaded program is one in which these are two parts of same program that can run concurrently.

16.4 Creating Thread Applications

There are two ways to create a thread application.

1. **By extending Thread class.**
2. **By implementing Runnable interface**

Thread Class.

Field Summary	
static int	**MAX_PRIORITY** The maximum priority that a thread can have.
static int	**MIN_PRIORITY** The minimum priority that a thread can have.
static int	**NORM_PRIORITY** The default priority that is assigned to a thread.

Constructor Summary
Thread() Allocates a new Thread object.
Thread(String name) Allocates a new Thread object with specified name.

Method Summary	
long	**getId**() Returns the identifier of this Thread.
String	**getName**() Returns this thread's name.
int	**getPriority**() Returns this thread's priority.
Thread.State	**getState**() Returns the state of this thread.
static boolean	**interrupted**() Tests whether the current thread has been interrupted.
boolean	**isAlive**() Tests if this thread is alive.
void	**join**() Waits for this thread to die.
void	**join**(long millis) Waits at most millis milliseconds for this thread to die.
void	**run**() If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.
void	**setName**(String name) Changes the name of this thread to be equal to the argument name.
void	**setPriority**(int newPriority) Changes the priority of this thread.
static void	**sleep**(long millis) Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

static void	**sleep**(long millis, int nanos) Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds.
void	**start**() Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
String	**toString**() Returns a string representation of this thread, including the thread's name, priority, and thread group.

1. By extending the Thread class

1. Create a sub class that extends **`Thread`** class.
2. Implement the **`run()`** method that contains logic of thread.
3. Create a object for newly created thread and then call the **`start()`** method which calls the **`run()`** method.

Program

```
class Mythread extends Thread
{
	public void run()
	{
		try
		{
			System.out.println("Welcome to Palar");
			Thread.sleep(2000); //wait for milli seconds
			System.out.println("Welcome to Java");
		}
		catch(Exception e)
		{
			System.out.println("Error");
		}
	}
}
class SingleThreadDemo
{
	public static void main(String args[])
	{
		// create object for thread
		Mythread t1= new Mythread();
		t1.setPriority(Thread.MAX_PRIORITY);
		t1.setName("Palar");
		System.out.println("Thread t1= "+t2);
		t1.start( );
	}
}
```

OUTPUT

```
		Welcome to Palar
			.....wait for 2000 miliseconds
		Welcome to Java
```

2. By Implementing Runnable Interface

1. Create a class that implements runnable interface.

2. Implement the run method that contains the logic of the thread.
3. Once the thread is created it can be started by the creating an instance and then calling start method.

```
class MyThread implements Runnable
{
      public void run( )
      {
            try
            {
            System.out.println("Welcome to Java");
            Thread.Sleep(3000);
            System.out.println("Welcome to Palar");
            }
            catch(Exception e)
            {
            System.out.println("Error");
            }
      }
  }
class RunnableDemo
{
  public static void main(String args[])
  {
      MyThread mt =new MyThread ( );
      Thread t = new Thread(mt);
      t.start();
  }
}
```

16.5 Creating Multi-threaded applications

```
class First extends Thread
{
      public void run( )
      {
            try
            {
                  System.out.println("Executing First thread");
                  Thread.sleep(1000);
                  System.out.println("End of  First thread");
            }
            catch(Exception e)
            {
                  System.out.println("Error");
            }
      }
}
      class Second extends Thread
      {
            public void run( )
            {
                  try
                  {
                        System.out.println("Executing second thread");
                        Thread.sleep(5000);
                        System.out.println("End of second thread");
                  }
                  catch(Exception e)
                  {
```

```
                        System.out.println("Error");
                }
        }
}

class Third extends Thread
{
        public void run( )
        {
                try
                {
                        System.out.println("Executing third thread");
                        Thread.sleep(2000);
                        System.out.println("End of third thread");
                }
                catch(Exception e)
                {
                        System.out.println("Error");
                }
        }
}
class MultiThread
{
        public static void main(String args[])
        {
                First t1=new First( );
                Second t2=new Second( );
                Third t3 =new Third( );
                t1.start( );
                t2.start( );
                t3.start( );
        }
}
```

OUTPUT

```
Executing First thread
Executing Second thread
Executing Third thread
End of first thread
End of third thread
End of second thread
```

16.6 join() method

The **`join()`** method makes caller wait till the current thread finishes execution.

```
class JoinDemo
{
        public static void main(String args[])
        {
                First t1=new First( );
                Second t2=new Second( );
                Third t3 =new Third( );
        try
        {
                t1.start( );
                t1.join( );
                t2.start( );
                t2.join( );
                t3.start( );
                t3.join( );
        }
```

```
            catch(Exception e)
            {
                  System.out.println("Error");
            }
            }
      }
```

OUTPUT

```
                  Executing first thread
                  End of first thread
                  Executing second thread
                  End of second thread
                  Executing Third thread
                  End of third Thread
```

In this case it waits until the first thread finishes, before starting the second thread.

16.7 isAlive() Method

It is used to determine whether a thread has finished. The **`isAlive()`** method returns true if it is still running otherwise it returns false.

```
class AliveDemo
{
      public static void main(String args[])
      {
                  First t1=new First( );
                  Second t2=new Second( );
                  Third t3 =new Third( );
      try
      {
            t1.start( );
            if(t1.isAlive())
            System.out.println("First Thread is running");
            t1.join( );
            if(t1.isAlive())
            System.out.println(t1.isAlive( ));
            else
            System.out.println("First Thread finished");
            t2.start( );

            t3.start( );

      }
      catch(Exception e)
      {
            System.out.println("Error");
      }
      }
}
```

OUTPUT

```
            Executing first thread
            true
            End of first thread
            False
            Executing second thread
            End of second thread
            Executing third thread
            End of third thread
```

16.8 Synchronization

When two or more threads need access to a shared resources, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is called ***synchronization.***

Program

```
class Callme
{
  synchronized void call(String msg)
 {
    System.out.print("[" + msg);
    try
       {
            Thread.sleep(1000);
       }
      catch(InterruptedException e)
      {
             System.out.println("Interrupted");
      }
    System.out.println(" ]");
  }
}

class Caller implements Runnable
{
  String msg;
  Callme target;
  Thread t;

  public Caller(Callme targ, String s)
{
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }

      public void run()
      {
          target.call(msg);
      }
}

class Synch
{

public static void main(String args[])
{
      Callme target = new Callme();
      Caller ob1 = new Caller(target, "Hello");
```

```
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");

    }
}
```