SRINIVASAN COLLEGE OF ARTS & SCIENCE

(Affiliated Bharathidasan University, Tiruchirappalli)

PERAMBALUR – 621 212

Department of Computer Science & Information Technology

Course Material

Subject	:	**Programming in JAVA**
Subject Code	:	**16SCCIT4**
Class	:	**II B.Sc IT**
Semester	:	**IV**

PROGRAMMING IN JAVA

Objective:

To understand the basic concepts of Object Oriented Programming with Java language

Unit I

Object Oriented Programming : Introduction to OOP – Objects and Classes – Characteristics of OOP – Difference between OOP and Procedure Oriented Language – Introduction to java Programming : Introduction – Features of Java – Comparing java and Other Languages – Applications and Applets – Java Development Kit – Complex Programs – Java Source File Structure – Prerequisites for Compiling and Running Java Programs

Unit II

Java Language Fundamentals : The Building Blocks of Java – Data Types – Variable Declarations – Wrapper Classes – Operations and Assignment – Control Structures – Arrays – Strings – StringBuffer Class

Unit III

Java as an OOP Language : Defining Classes – Modifiers – Packages - Interfaces

Unit IV

Exception Handling : Introduction – Basics of Exception Handling – Exception Hierarchy – Constructors and Methods in Throwable Class - Unchecked and Checked Exceptions – Handling Exceptions in Java – Exception and Inheritance – Throwing User-defined Exceptions – Redirecting and Rethrowing Exceptions – Advantages of Exception Handling Mechanism – Multithreading : Introduction – Creating Threads – Thread Life-cycle – Thread Priorities and Thread Scheduling – Thread Synchronization – Daemon Threads – Tread Groups – Communication of Threads

Unit V

Files and I/O Streams : Overview – Java I/O – File Streams – FileInputStream and FileOutputStream – File Streams – RandomAccess File – Serialization - Applets : Introduction – Java Applications versus Java Applets – Applet Life-cycle – Working with Applets – The HTML APPLET Tag – The java.Applet package

Text Book :

1. Object Oriented Programming through Java, P. Radha Krishna, University Press, 2011.

Reference Book:

1. Java Programming, K. Rajkumar, Pearson India, 2013

PROGRAMMING IN JAVA

Fundamentals of Object Oriented Programming:

Many programming models have evolved in the domain of software solutions. The main models are

1. **Procedural Programming Model** : Each problem is divided into smaller problems and solved using specified modules that act on data.

2. **Object oriented programming model:** It perceived the entire software system as a collections of objects which contain attributes and behaviors.

Object:

An object is a tangible entity that may exhibit some well-defined behavior.
(Or) Object is an instance of Class (or) Everything is an object.

Class:

A class is a se of attributes and behaviors shared by similar objects.
(or) in simple way Collection of objects is called Class.

Abstraction:

Abstraction focuses on the essential characteristics of an objects.

Encapsulation :

Encapsulation hides the implementation details of an object and thereby , hides its complexity.

Inheritance :

Inheritance creates a hierarchy of classes and helps in the reuse of the attributes and methods of a class.

Polymorphism:

Polymorphism triggers different reactions in objects in response to the same message.(or) More than one function in same name.

Ex:

```
Public void abc (int a , int b)
        {

        }
public void abc (int x, int y, int z)
        {

        }
```

Super class :

A super class shares its attributes and behavior with its child classes.

Sub class:

A subclass inherits attributes and behaviors from its super class.

Abstract class:

An abstract class is a conceptual class. The objects of an abstract class do not exist in the real world.

HISTORY OF JAVA:

IN 1991, SUN Microsystems began a project called "Green " to develop software for use in consumer electronics. The leader of the project James Gosling and include Patrick Naughton, Mike Sheridan. The team wanted a fundamentally new way of computing, based on the power of networks , and wanted the same software to run on different kinds of computers and different operating system. They create new language and give the name OAK.

In this time the WWW (World Wide Web) was in a period of dramatic growth , Sun realized that Oak was perfectly suited for Internet process. In 1994 , they completed work on a product known as a Webrunner. Later renamed as HotJava. Hotjava demonstrated the power of Oak as Internet development tools.

In 1995 , OAK was renamed as Java (Marketing purpose).

FEATURES OF JAVA :

Java is a simple language that can be learned easily, even if you have just started programming.

A java programmer need not know the internals of java. The syntax of Java is similar to C++. Unlike C++ in which the programmer handles memory manipulation, Java handles the required memory manipulation and thus prevents errors that arise due to improper memory usage.

Java is purely object-oriented and provides ***abstraction, encapsulation, inheritance*** , and ***polymorphism.*** Even the most basic program ha a class. Any code that you write in Java is inside a class.

Java is tuned to the Web. Java programs can access data across the Web as easily as they access data from a local system. You can build ***Distributed*** applications in Java that use resources from any other networked computer.

Java is both ***Interpreted*** and ***Compiled***. The code is compiled to a ***bytecode*** that is ***binary*** and ***platform-independent.***

When you compile a piece of code, all the errors are listed together. You can execute a program only when all the errors have been rectified. Only when the execution reaches the statement with an error is the error reported. This makes it easy for a programmer to debug the code.

A programmer cannot write Java code that interacts with the memory of the system. Instead of assuming that programmers know what they are doing, java ensures the same. For instance , Java forces you to handle unexpected errors. This ensures that Java programs are ***robust***(reliable), and bug free and do not crash.

A program traveling across the Internet to your machine could possibly be carrying a virus. Due to strong type-checking done by Java on the user's machine, any changes to the program are tagged as errors and the program will not execute. Java is, therefore, ***secure***.

Java programs are comparable in speed to the programs written in other compiler-based languages like C and C++. Java is ***faster*** than other interpreter-based languages like BASIC since it is compiled and interpreted.

Multithreading is the ability of an application to perform multiple tasks at the same time. For example, when you play a game on your computer, one task of the program is to handle sound effects and another to handle screen display. A single program accomplishes many tasks simultaneously. Microsoft Word is another multithreaded program in which the data automatically saved as you key it in. You can create multithreaded programs using Java. The core of Java is also multithreaded.

A Java program can consist of many modules that are written by many programmers. These modules may undergo many changes. Java makes interconnections between modules at run-time, which easily avoids the problems caused by the change of the code used by your program. Java is thus ***dynamic.***

The following definition of Java by Sun Microsystems lists all the features of Java.

" ***java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral ,portable, high-performance, mulththreaded, and dynamic language".***

JDK TOOLS:

Java Development ToolKit(JDK) is a software package form Sun Microsystems. Version 1.1 was released with major revisions to the original version(1.0). The latest version of JDK is JDK 1.3.

The javac Compiler:

You can create Java programs using any text editor. The file you create should have the extension .java. Every file that you create (source code) can have a maximum of one *public* definition. The source code is converted to byte code by the *javac* compiler converts the .java file that you create to a .class file, which contains byte code.

Syntax for compiling java code using javac:

Javac<filename.java>

The java Interpreter

The java interpreter is used to execute compiled java applications. The byte code that is the result of compilation is interpreted so that it can be executed.

Syntax for executing a java application using java:

Java<filename.class>

- open a new file in the editor and type the following script

```
class first
{
public static void main(String args[])
{
System.out.println(“This is a simple java program”);
}
}
```

- save file as **first.java**
- compile by typing **javac first.java** on the command line.
- On successful compilation execute the program by typing **java first** on the command line
- The program display **This is a simple java program** on the screen.

In this program **class** to declare that a new class is being defined. **First** is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace({) and the closing curly brace (}).

Next line begins the **main( )** method. As the comment preceding it suggests, this the line at which the program will begin executing. All java applications begin execution by calling **main( )**.

The **public** keyword is an access specifier, which allows the programmer to control the visibility of class member. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared.

The **static** allows main( ) to be called without having to instantiate a particular instance of objects are made.

The keyword **void** simply tells the compiler that main( ) does not return a value.

String args[ ] declares a parameter named **args,** which is an array of instances of the class **String.** (Arrays are collections of similar objects.). objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed.

Next **System.out.println( );** Output is actually accomplished by the built-in **println( )** method. In this case, **println( )** displays the string which is passed to it. **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

Data Types:

The data that is stored in memory can be of many types. For example, a person's age is stored as a numeric value and an address is stored as alphanumeric characters. Data types are used to define the operations possible on them and the storage method.

The data types in Java are classified as

* ***Primitive or Standard data types***

**** Abstract or derived data types.***

Standard Data Types

Integer:

Integers are used for storing integer values. There are four kinds of integer types in Java. Each of these can hold a different range of values. The values can either be positive or negative.

Type	Size/format	Range
Byte	8 bit	-128 to +127
Short	16 bit	-32,768 to + 32,767 (-2^{15} to $2^{15}-1$)
Int	32 bit	-2,147,483,648 to +2,147,483,647
Long	64 bit	-9223372036854775808 to +9223372036854775807 (-2^{63} to $2^{63}-1$)

Float:

Float is used to store numbers with decimal part. There are two floating point data types in Java .

Type	Size/format	Range
Float	32 bit	+/- about 10^{39}
double	64 bit	+/- about 10^{317}

Character:

It is used for storing individual characters.

char 16 bits of precision and it is unsigned.

Boolean:

Boolean data types hold either a true or a false value.

boolean 1 bit (*true or false*)

Derived Data Types:

Abstract data types are base on primitive data types and have more functionality than primitive data types. For example, String is an abstract data type that can store letters, digits and other characters like /,( ) : ; $ and #.

String provides methods for concatenating two strings , searching for one string within another, and extracting a portion of a string. The standard data types do not have these features.

VARIABLES:

Variables are locations in the memory that can hold values. Before assigning any value to a variable, it must be declared. To use the variable *number* storing an integer value , the variable *number* must be declared and it should be of the type *int.*

Rules for Naming Variable:

The name of a variable needs to be meaningful, short, and without any embedded space or symbol like - ? @ # % ^&*( ) [ ] . , ; ; “ ‘ / and \ . However , underscores can be used wherever a space is required; for example basic salary.

Variable names must be unique. For example, to store four different numbers, four unique variable names need to be used.

A variable name must begin with a letter , a dollar symbol($) or an underscore(_) , which may be followed by a sequence of letters or digits (0 – 9) ,’&’ or ‘_’.

Keywords cannot be used for variable names. For example , you cannot declare a variable called *switch.*

LITERALS:

The variables in Java can be assigned constant values. The values assigned must match the data type of the variables. Literals are the values that may be assigned to primitive or string type variables and constants.

The Boolean literals are *true* and *false.*

The integer literals are numeric data. Ex(45)

The floating-point literals are numbers that have decimal fraction. Ex(34.5)

Character literals are enclosed in single quotes ex('a').

String literals are enclosed in double quotes ex ("soft").

Escape sequence character:

\n new line . \t tab. \b backspace.

OPERATORS:

Operators are used to compute and compare values, and test multiple conditions. They classified are Arithmetic Operator, Assignment Operator , Unary Operators, Comparison Operators, Shift Operators, Bit-Wise Operators , Logical Operators, Conditional Operators, new Operator.

Arithmetic Operators:

Operator	Description	Example	Explanation
+	Adds the operands	X=y + z	Adds the value of y and z and stores the result in x.
-	Subtracts the right operand form the left operand	X = y – z	Subtracts z from y and stores the result in x.
*	Multiplies the operands	X = y * z	Multiples the values y and z and store the result in x.
/	Divides the left operand by the right operand.	X = y / z	Divides y by z and stores the result in x.
%	Calculates the remainder of an integer division.	X = y % z	Divides y by z and stores the remainder in x.

The + operator with Numeric data types:

When you add two operands of the primitive numeric data type, the result is a primitive numeric data type.

```
byte soft = 100;
byte arun= 20;
byte simi=soft+arun;
```

Then print the simi value is 120.

The + operator with String Data types:

When you use the + operator with numeric operands, the result is numeric. When both the operands are strings, the + operator concatenates(joins) them. When one of the operands is a String object, the second operand is converted to String before concatenation. For ex.

Operand1	Operand2	Result
5	6	11
5	“soft”	“5soft”
“soft”	“arun”	“softarun”

ASSIGNMENT OPERATOR:

Operator	Description	Example	Explanation
=	Assigns the value of the right operand to the left	X=40	Assigns the value of 40 to x
-=	Subtracts the right operand form the left operand and stores the result in the left operand.	x-=y	Subtracts y from x. x= x – y
+=	Adds the operands and assigns the result to the left operand.	X+= y	Adds the value of y to x. X= x + y
*=	Multiplies the left operand by the right operand and stores the result in left operand	X*=y	Multiplies the values x and y and stores the result in x. x = x * y.
/=	Divides the left operand by the right operand and stores the result	X/=y	Divides x by y and stores the result in x. x = x / y
%=	Divides the left operand by the right operand and stores the remainder in the left operand	X%=y	Divides x by y and stores the remainder in x. x = x % y.

UNARY OPERATORS			
++	Increases the value of the operand by one	X++	Equivalent to x = x + 1
--	Decrease the value of the operand by one	X--	Equivalent to x = x – 1.
Ex : **Pre-increment** a=1; b=++a; After executing b=2.		**Post-increment** c=1; d=c++; After executing d=1.	
COMPARISON (RELATIONAL) OPERATORS			
= =	Evaluates whether the operands are equal	X = = y	Returns *true* if the values are equal otherwise *false.*
!=	Evaluates whether the operands are not equal	X != y	Returns *true* if the values are not equal otherwise *false.*
>	Evaluates whether the left operand is greater than the right operand	X > y	Returns *true* if x is greater than y otherwise *false*..
<	Evaluates whether the left operand is less than the right operand.	X < y	Returns *true* if x is less than y otherwise *false.*
>=	Evaluates whether the left operand is greater than or equal to the right operand	X >= y	Returns *true* if x is greater than or equal to y otherwise *false.*
<=	Evaluates whether the left operand is less than or equal to the right operand	X <= y	Returns *true* if x is less than or equal to y otherwise *false.*

BIT-WISE OPERATORS

Operator	**Description**	**Example**	**Explanation.**
& (AND)	Evaluates to a binary value after a bit wise AND on the operand	X & Y	AND results in a 1 if both the bits are 1. any other combination results in a 0.
| (OR)	Evaluates to a binary value after a bit wise OR on the two operand	X | Y	OR results in a 0 if the both the bits are 0. any other combination results in a 1.
^ (XOR)	Evaluates to a binary value after a bit wise XOR on the two operand.	X ^ Y	XOR results in a 0 if both the bits are of the same value and 1if the bits have diff. values.
~	Converts all 1 bits to 0s and all 0s bits to 1s		Example given bellow
			Example: a=1010001 a~ = 0101110

LOGICAL OPERATORS			
&&	Evaluates to *true* if both the condition evaluate to *true* other wise *false.*	X>5 && Y <5	The result is *true* if condition1(x>5) and condition2 (y<5) are both true. If one of them *false* , the result is *false*.
||	Evaluates to *true* if at least one of the conditions evaluates to *true,* and *false* if none of the conditions evaluates to *true.*	X > 5 || y <10	The result is *true* if either condition1(x>5) or condition2 (y<10) or both evaluate to *true.* If both the conditions are *false,* the result is *false*
CONDITIONAL OPERATOR			
(Condition) ? val1 : val2	Evaluates to val1 if the condition returns true and val2 if the condition returns false	**X = (y>z) ? y :z**	X is assigned the value of y if y is greater than z , else x is assigned the value of z.
SHIFT OPERATORS			
>>	Shifts bits to the right, filling sign bits at the left and it is also called the signed right shift operator	X=10 >>3	The result of this is 10 divided by 2^3. an explanation is given bellow.
<<	Shifts bits to the left filling zeros at the right.	X=10<<3	Result of this is 10 multiplied by 2^3.
>>>	Also called the unsigned shift operator works like the >> operator, but fills in zeroes for the left.	X= -10 >>> 3	An explanation is given bellow.

The int data type occupies four bytes in the memory. The rightmost eight bits of the number 10 are represented in binary as

0	0	0	0	1	0	1	0

When you do a right shift by 3 (10 >>3), the result is $10/2^3$

0	**0**	**0**	0	0	0	0	1

When you do a left shift by 3 (10 <<3) , the result is $10 * 2^3$ which is equivalent to 80.

0	1	0	1	0	**0**	**0**	**0**

The new Operator:

When you create an instance of a class, you need to allocate memory for it. When you declare an object, you merely state its data type.

Pen blcakPen; This tells the compiler that the variable *blackPen* is an object of the *Pen* class. It does not allocate the memory for the object.

To allocate the memory, you need to use the *new* operator.

Syntax:

<class_name> = new<class_name>( );

Example:

Pen blockPen= new Pen( );

ORDER OF PRECEDENCE OF OPERATORS:

[ ] , ( ) , + , - , ~ , !, ++ , -- , *,/,%,+,-,

<<,>>,>>>,<,<=,>=,>,= =,!=,

&,^,|,&&,||,?:,=,+=,-=,*=,/=,%=

CONTROL STATEMENTS:

if .. else statements.

The **if** decision construct is followed by a logical expression in which data is compared and a decision is made based on the result of comparison. The condition is true then true part statement is to be executed and exit the loop otherwise else part statement is to be executed and exit the loop.

Syntax:

```
if (boolean_expr)
{
        statements;
}
else
{
        statements;
}
```

example:

```
class ifodd
{
public static void main(String args[])
{
int n=Integer.parseInt(args [0]);
if(n%2 ==0)
System.out.println("Given number " + n +"is even");
else
System.out.println("Given number " + n + "is odd ");
}
}
```

while Loop statements.

The **while** loop is a looping construct available in java. The **while** loop continues until the evaluating condition becomes **false**. The evaluating condition has to be a logical expression and must return a **true** or **false** value. The variable that is checked in the Boolean expression is called the **loop control variable.**

Syntax:

```
While(Boolean_expr)
{
        statements
}
```

example:

```
class facwhile
{
public static void main(String args[])
{
int n=Integer.parseInt(args [0]);
int i=1, f=1;
while(i<=n)
{
f=f*i;
i=i+1;
}
System.out.println(+f);
 }}
```

do .. while Loop:

In a while loop, the condition is evaluated at the beginning of the loop. If the condition is **false**, the body of the loop is not executed. If the body of the loop must be executed at least once, than the **do..while** construct should be used. The **do .. while** construct places the test expression at the end of the loop.

The keyword **do** marks the beginning of the loop. The braces delimit the body of the loop. Finally, a **while** statement provides the condition and ends the body of the loop.

Syntax:

```
do
{
        statements;
}while(boolean_expr);
```

example:

```
class arms
{
public static void main(String args[])
{
int n=Integer.parseInt(args[0]);
int s=0,t=n,r;
do
{
r=n%10;
s=s+r*r*r;
n=n/10;
}while(n!=0);
if(t==n)
System.out.println("Given number is arms"+t);
else
System.out.println("Given number is not arms"+t);
}}
```

for Loop:

The while and the do..while loops are used when the number of iterations(the number of the times the loop body is executed) is not known. The **for** loop is used in situations when the number of iterations is known in advance. For example, it can be used to determine the square of each of the first ten numbers.

The **for** statement consists of the keyword **for**, followed by parentheses containing three expressions each separated by a semicolon. These are the **initialization** expression, the **test** expression and the **increment/decrement** expression.

Syntax:

```
for(initialization_expr;test_expr;increment/decrement_expr)
{
        statements;
}
```

Initialization expression:

The initialization expression is executed only once, when the control is passed to the loop for the first time. It gives the loop variable an initial value.

Test expression:

The condition is executed each time the control passes to the beginning of the loop. The body of the loop is executed only after the condition has been checked. If the condition evaluated to **true**, the loop is executed otherwise, the control passes to the statement following the body of the loop.

Increment / Decrement expression:

The increment / decrement expression is always executed when the control returns to the beginning of the loop.

example:

```
class fib
{
public static void main (String args[])
{
int n=Integer.parseInt(args[0]);
int n1=-1, n2=1,int n3;
System.out.println("FIBONACCI SERIES ");
for(int i=1;i<n;i++)
{
n3=n1+n2;
System.out.println(n3);
n1=n2;
n2=n3;
}}}
```

switch statement:

The **switch** statement is Java's multi way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements .

Syntax:

```
switch(expression)
{
case value1:
        stt.
        break;
case value2:
        Stt.
        break;
case value n:
        Stt.
        break;
default:        stt.
}
```

The *expression* must be of type **byte, short, int, or char;** each of the *values* specified in the **case** statements must be of a type compatible with the expression. Each **case** value must be a unique literal(that is, it must be a constant, not a variable). Duplicate **case** values are not allowed.

The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken.

The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has effect of "jumping out" of the **switch.**

Arrays:

An *array* is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions.

One-Dimensional Arrays:

An **one-dimensional array** is , essentially, a list of like-typed variables. To create a array, you first must create an array variable of the desired type. The general form of a one-dimensional array declaration is

Type var-name[ ];

The general form of **new** as it applies to one-dimensional arrays appears as follows

Array-var = new type [size];

Here *type* declares the base type of the array. The base type determines the data type determines the data type of each element that comprises the array.

Example

```
class sort
{
public static void main(String args[])
{
int a[]=new int[10];
int j,i,n;
n=Integer.parseInt(args[0]);
for(i=1;i<=n;i++)
{
a[i]=Integer.parseInt(args[i]);
}
for(i=1;i<=n;i++)
{
```

```
for(j=i+1;j<=n;j++)
{
if(a[i]>a[j])
{
int temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}}
for(i=1;i<=n;i++)
{
System.out.println(a[i]);
}
}}
```

Multidimensional Arrays:

In java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets.

A *two-dimensional* array can be thought of as a table of rows and columns.

Syntax:

Data type array-name[][];
(Or)
data type array-name[][]=new data type [row size] [column size];

Example:

```
class mattra
{
public static void main (String args[])
{
int a[][]=new int[10][10];
int b[][]=new int[10][10];
int j,i,k=0;
```

```
    for(i=1;i<=2;i++)
    {
    for(j=1;j<=2;j++)
    {
    a[i][j]=Integer.parseInt(args[k]);
    k=k+1;
    }
}
    for(i=1;i<=2;i++)
    {
    for(j=1;j<=2;j++)
    {
    b[i][j]=a[j][i];
    k=k+1;
    }}
    for(i=1;i<=2;i++)
    {
    for(j=1;j<=2;j++)
    {
    System.out.print("    "+ b[i][j]);
    }
    System.out.println(" ");
    }
    }
    }
```

JAVA AS AN OOP LANGUAGE

Java is a true object-oriented language and therefore the underlying structure of all java programs is classes. Anything we wish to represent in a java program must be encapsulated in a class that defines the ***state***and ***behavio****r* of the basic program components known as ***objects***. Classes create objects and objects use methods to communicate between them. That is all about object-oriented programming.

Classes provide a convenient method for packing together a group of logically related data items and functions that work on them. In java, the data items are called ***fields*** and the functions are called ***methods***. Calling a specific method in an object is described as sending the object a message

A class is essentially a description of how to make an object that contains fields and methods. It provides a sort of ***template***for an object and behaves like a basic data type such as **int**. it is therefore important to understand how the fields and methods are defined in a classes and how they are used to build a java program that incorporates the basic OOP concepts such as ***encapsulation, inheritance*** and ***polymorphism***.

Class :

Class is collection of objects.

Object:

Object is a instance of a particular class.

General form of class:

class class name
{
type instance variable 1;
type instance variable 2;
….
….
….
type instance variable n;

```
class name1(parameter-list)
{
body of constructor;
}

class name2(parameter-list)
{
body of constructor;
}
class name n(parameter-list)
{
body of constructor;
}

type methodname1(parameter-list)
{
body of method;
}

type methodname2(parameter-list)
{
body of method;
}

…..
type method name n(parameter-list)
{
body of method;
}
}
```

Instance variable:

Data or Variable declare within the class is called instance variable.

Method:

The source code within the class is called method.

Member:

The methods and variables defined within a class are called members of the class.

Creating a class:

```
class sample
{
int a;
int b;
int c;
}
```

This declaration defines a class called **sample** that consists of three integer members: **a , b and c.** It is important to understand that this declaration does not actually create any objects.

Creating objects:

An object in java is essentially a block of memory that contains space to store all the instance variables. Creating an object is also referred to as *instantiating* an object.

Objects in java are created using the **new** operator. The **new** operator dynamically allocates(that is , allocates run time)memory for an object and returns a reference to that object.

To create objects of type **sample**, use statements such as the following:

```
sample one = new sample( );
sample two = new sample( );
```

After these statements execute, there are two objects that have the form described by **sample.** The variable **one** holds a reference to one of these objects. The variable **two** holds a reference to the other. Each object has its own copies of variables **a, b, and c.**

Dot Operator:

The dot notation is used to obtain the value of the instance variables. It has two parts namely the object on the left side of the dot and the variable on the right side of the dot. Dot expressions are evaluated from left to right. The general form for accessing instance variables using the dot operator is given below:

Objectreference.variable name

We can store values into instance variables using the dot operator as shown below:

one.a=10; one.b=20; one.c=30;

two.a=15; two.b=10; two.c=35;

We can refer to values of instance variables using the dot operator as given below:

System.out.println(“a=”+one.a +”b=” +one.b +”c=”+one.c);

System.out.println(“a=”+two.a +”b=” +two.b +”c=”+two.c);

example for class and object:

This program save filename is excla this is the main class name.

```
class add
{
double a,b,c;
}
class excla
{
public static void main(String args[])
{
add one=new add();
double val;
one.a=10;
one.b=20;
one.c=30;
val=one.a+one.b+one.c;
System.out.println(val);
}}
```

METHODS:

Methods are functions that operate on instances of classes in which they are defined. Objects can communicate with each other using methods and can call methods in other classes.

Defining Methods:

Method definition has four parts. They are, name of the method, type of object or primitive type the method returns, a list of parameters and body of the method. A method's signature is a combination of the first three parts mentioned above. Java permits different methods to have the same name as long as the argument list is different. This is called method overloading. This is the general form of a method

```
type name(parameter-list)
{
        body of method;
}
```

here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void.** The name of the method is specified be *name.* The *parameter-list* is a sequence of type and identifier pairs separated by commas, the method when it is called. If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than **void** return a value to the calling routine using the following form of the **return** statement: return *value;*

Here , *value* is the value returned.

Calling Methods:

Calling a method is similar to calling or referring to an instance variable. The methods are accessed using the dot notation. The object whose method is called is on the left of the dot while the name of the method and its arguments are on the right.

obj. method name(parameter1, parameter2);

example for method:

```
class exmet
{
int a,b,c;
void cal( )
{
System.out.println("example of method ");
System.out.println(a*b*c);
}
public static void main(String args[])
{
exmet mymul=new exmet( );
mymul.a=Integer.parseInt(args[0]);
mymul.b=20;
mymul.c=15;
mymul.cal( );
}}
```

passing Argument to methods:

The objects that are passed to the body of the method are passed by reference and the basic types are passed by value. This results in a change in original value of the object if the valued is modified in the method.

Example the passing of arguments to methods.

```
class mul
{
int a;
int sqr(int a)
{
System.out.println("squre value is ");
```

```
return a*a;
}}
class exmet3
{
public static void main(String args[])
{
mul mymul=new mul();
int vol;
 vol=mymul.sqr(5);
System.out.println(vol);
}}
```

The this keyword:

A special reference value called **this** is included in java. The **this** keyword is used inside any instance method to refer to the current object. The value **this** refers to the object which the current method has been called on. The **this** keyword can be used where a references to an object of the current class type is required.

Example of this keyword:

```
class this1
{
int x,y;
void show(int x,int y)
{
this.x=x; this.y=y;
}
void disp()
{
System.out.println("x="+x);
System.out.println("y="+y);
}}
class exthis
{
public static void maim (String args[])
{
this1 th=new this1();
th.show(4,6);
th.disp();
}}
```

output of the program is : x=4 y=6.

Overloading Methods:

In java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloade*d, and the process is referred to as *method overloading.* When we call a method in an object, java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as *polymorphism.*

Example of overloading method:

```
class exover
{
void test()
{
System.out.println("No parameters");
}
void test(int a)
{
System.out.println("a :"+a);
}
void test(int a, int b)
{
System.out.println("a and b "+a+ "  "+b);
}
double test(double a)
{
System.out.println("double value is "+a);
return a*a;
}
public static void main(String args[])
{
exover ex=new exover();
double d;
ex.test();
ex.test(10);
ex.test(10,20);
d=ex.test(120.5);
System.out.println("double result is "+d);
}}
```

Constructors:

Often an object will require some form of initialization when it is created. To accommodate this, java allows you to define constructors for your classes. A *constructor* is a special method that creates and initializes an object of a particular class. It has the same name as its class and may accept arguments. In this respect, it is similar to any other method.

However, a constructor does not have a return type. Instead, a constructor returns a reference to the object that it creates. If you do not explicitly declare a constructor for a class, the java compiler automatically generates a default constructor that has no arguments.

A constructor is never called directly. Instead, it is invoked via the **new** operator.
And allocate the memory space.

example for constructor :

```
class mul
{
double a,b,c;
mul()
{
System.out.println("Constructor example");
a=10;
b=5;
c=10;
}
double cal()
{
System.out.println("calculated value is ");
return a*b*c;
}
}
class excon
{
public static void main(String args[])
{
mul mymul=new mul();
double vol;
vol=mymul.cal();
System.out.println(vol);
}}
```

Constructor Overloading:

A class may have several constructors. This feature is called constructor overloading. When constructors are overloaded, each still called by the name of its class. However, it must have different parameter list. In more precise terms, the signature of each constructor must differ.

example for constructor overloading:

```
class exover1
{
int a,b,c;
exover1(int x,int y,int z)
{
a=x;b=y;c=z;
}
exover1()
{
a=1;b=1;c=1;
}
exover1(int d)
{
a=d;b=d;c=d;
}
int cal()
{
return a*b*c;
}
public static void main(String args[])
{
exover1 ex=new exover1(5,5,5);
exover1 ex1=new exover1();
exover1 ex2=new exover1(2);
int k;
System.out.println("first resutl");
k=ex.cal();
System.out.println(k);
System.out.println("second result");
k=ex1.cal();
System.out.println(k);
System.out.println("third result");
k=ex2.cal();
System.out.println(k);
}}
```

Recursion:

Java supports *recursion.* The method call by itself is called recursive method.

Example for recusion:

```
class fact
{
int fact(int n)
{
int result;
if(n==1) return 1;
result=fact(n-1)*n;
return result;
}
public static void main(String args[])
{
fact f=new fact();
System.out.println("factorial value is "+f.fact(4));
System.out.println("factorial value is "+f.fact(5));
System.out.println("factorial value is "+f.fact(6));
}}
```

Inner Classes:

It is possible to nest a class definition within another class and treat the nested class like any other method of that class. Such a class is called nested class. As a member of its enclosing class , a nested class has privileges to access all the members of the class enclosing it. A nested class can either be static or not-static. While static nested classes are just called as static nested classes, non-static classes are called as inner classes.

Example for inner classes:

```
class inner
{
void test()
{
inn in=new inn();
in.disp();
}
class inn
{
int x=10,y=5;
```

```
void disp()
{
int z=x+y;
System.out.println("example for inner class ");
System.out.println("x + y value is : "+z);
}}

public static void main(String args[])
{
inner o=new inner();
o.test();
}}
```

Inheritance:

Inheritance is new class derived from old class. Some modification about particular class. It should invoke super class(base class) & sub class(derived class).

Types of inheritance:

1. Single Inheritance (only one super class).
2. Multiple Inheritance (several super class).
3. Multilevel Inheritance (derived from a derived class).
4. Hybrid Inheritance (combination of multiple and multilevel inheritance).
5. Hierarchical Inheritance (one super class, many subclasses).

Java does not directly implement multiple inheritance. However, this concept is implemented using a secondary inheritance path in the for of *interfaces.*

Defining a subclass:

```
class subclass name extends super class name
{
        variable declaration;
        methods declaration;
}
```

The keyword **extends** signifies that the properties of the **super class name** are extended to the **subclass name.** The subclass will now contain its own variables and methods as well those of the super class. This kind of situation occurs when we want to add some more properties to an existing class without actually modifying it.

Example for single inheritance:

```
class a
{
int i;
private int j;
void setval(int x,int y)
{
i=x;j=y;
System.out.println("i value is "+i  +"   j value is "+j);
}
}
class b extends a
{
int total,j=15;
void sum()
{
total =i+j;
System.out.println("total is :"+total);
}
}
class inher1
{
public static void main(String args[])
{
b bb=new b();
bb.setval(4,6);
bb.sum();
}
}
```

example multilevel inheritance

```
class a
{
int i;
private int j;
void setval(int x,int y)
{
i=x;j=y;
System.out.println("i and j value is "+i  +" "+j);
}}
class b extends a
{
int total,k=10;
void sum()
{
total =i+k;
System.out.println("first subclass value is "+total);
}}
class c extends b
{
int a,b,c;
void cal(int a)
{
c=a+i+k;
System.out.println("multilevel inheritance is "+c);
}
void setval(int x)
{
b=x;
System.out.println("b value is "+b);
}
}
class inher2
{
public static void main(String args[])
{
c bb=new c();
bb.setval(10,6);
bb.setval(5);
bb.sum();
bb.cal(100);
}}
```

Super keyword:

A subclass constructor is used to construct the instance variable of both the subclass and the superclass. The subclass constructor uses the keyword **super** to invoke the constructor method of the super class. The keyword **super** is used subject to the following conditions

- **Super** may only be used within a subclass constructor method.
- The call to super class constructor must appear as the first statement within the super class constructor.
- The parameters in the **super** call must match the order and type of the instance variable declared in the super class.

Example for super keyword:

```
class a
{
int i;
}
class b extends a
{
int i;
b(int a,int b)
{
super.i=a;
i=b;
}
void show()
{
System.out.println("base class variable i value is " +super.i);
System.out.println("sub class variable i value is " +i);
}}
class exsuper
{
public static void main(String args[])
{
b ob=new b(4,6);
ob.show();
}}
```

Method Overriding:

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.

Example for overriding:

```
class a
{
void hello( )
{
System.out.println(“hello from a”);
}}



class b extends a
{
void hello( )
{
System.out.println(“hello from b”);
}}
class c extends b
{
void hello( )
{
System.out.println(“hello from c”);
}}
class exover
{
public static void main(String args[])
{
c ob=new c( );
ob.hello( );
}}
```

output from this application is shown here:

hello from c

another example for overriding:

```
class a
{
int i,j;
a(int a,int b)
{
i=a;j=b;
}
void show()
{
System.out.println("i value is "+i);
System.out.println("j value is "+j);
}}
class b extends a
{
int k;
b(int a,int b,int c)
{
super(a,b);
k=c;
}

void show()
{
super.i=10;
System.out.println("k value is "+k);
super.show();
}
}
class supers
{
public static void main(String args[])
{
b ob=new b(1,2,3);
ob.show();
}}
```

Hierarchical inheritance:

It is one base class many sub class.

Example of Hierarchical inheritance:

```
class a
{
int i;
 int j;
void setval(int x,int y)
{
i=x;
j=y;
System.out.println("i and j value is "+i  +" "+j);
}}
class b extends a
{
int total;
void sum()
{
total =i+j;
System.out.println("first subclass add value is "+total);
}}
class c extends a
{
int c;
void cal()
{
System.out.println(" i value is "+i);
System.out.println(" j value is "+j);
c=i-j;
System.out.println("Second subclass sub vlaue is "+c);
}}

class inher3
{
public static void main(String args[])
{
b bb=new b();
bb.setval(10,6);
bb.sum();
c cc=new c();
cc.setval(15,6);
cc.cal();
}}
```

In this program save as file name is inher3. This program base class is “a” . Then “b” and “c “ class are the subclass of “a” class. These two class are different operation this program to satisfy the one base class more than one sub class so it’s hierarchical inheritance program.

Interfaces:

Using the keyword ***interface***you can fully abstract a class interface from its implementation. That is, using **interface** you can specify what a class must do, but not how it does it.

Like classes, interfaces contain methods and variables but with a major difference. The difference is that interfaces define only abstract methods and final fields. This means that interfaces do not specify and code to implement these methods and data fields contain only constants.

Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods.

The syntax for defining an interface is very similar to that for defining a class. The general form of an interface definition is:

interface Interfacename
{
    variable declaration;
    methods declaration;
}

Here, **interface** is the key word and Interface name is any valid java variable(just like class names). Variables are declared as follows:

Type final-variable-name1=value;
Type final-variable-name2=value;

Note that all variables are declared as constants. Methods declaration will contain only a list of methods without any body statements. Example

return-type methodname1(parameter-list);
return-type methodname2(parameter-list);

Here is an example of an interface definition that contains two variables and one method

```
interface area
    {
        final float pi=3.142f;
        float compute (float x, float y);
        void show( );
    }
```

here pi is the float type variable it will be declare as final suppose you declare simply float pi=3.142f that will be consider as final type.

Extending interfaces:

Like classes, interfaces can also be extended. That is , an interface can be sub interfaced from other interfaces. The new sub interface will inhert all the members of the super interface in the manner similar to subclasses. This is achieved using the keyword **extends** as shown below:

```
Interface name2 extends name1
{
    body of name2;
}
```

Example:

```
interface itemconstant
{
    int code= 1001;
    String name=”fan”;
}
interface item extends itemconstant
{
    void display( );
}
```

while interfaces are allowed to extend to other interfaces, sub interfaces cannot define the methods declared in the super interfaces. After all, sub interfaces are still interfaces, not classes. Note that when an interface extends two or more interfaces, they are separated by commas.

It is important to remember that an interface cannot extend classes. This would violate the rule that an interface can have only abstract methods and constants.

Implementing interface:

Interfaces are used as “super classes” whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

Class classname **implements** interfacename
{
    body of classname;
}

Here the class classname “implements” the interface interfacename. A more general form of implementation may look like this:

Class classname **extends** superclass **implements** interface1,interface2,…
{
    body of classname;
}

This is shows that a class can extend another class while implementing interfaces.

When a class implements more than one interface, they are separated by a comma.

Example of Interface with implements (Multiple inheritance)

```
class onee
{
int a,b,c;
void setval(int x,int y)
{
a=x;
b=y;
}

void cal()
{
c=a+b;
System.out.println("base class 1 " + c);
```

```
}
}

interface two
{
int x = 10;
}

class three extends onee implements two
{
int y= 20;
void cal1()
{
c=a+x+y;
System.out.println("multiple inheritance c value is"+c);
}
}

class mulpinh
{
public static void main(String args[])
{
three t=new three();
t.setval(4,6);
t.cal();
t.cal1();
}
}
```

Hybrid inheritance:

Combination of multiple and multiple inheritance is called hybrid inheritance.

Example of hybrid inheritance is:

```
class student
{
int rollno;
void getnumber(int n)
{
rollno=n;
}
void putnumber()
{
System.out.println("Roll Number :"+rollno);
}
}
class mark extends student
{
int ma1,ma2;
void getmarks(int m1,int m2)
{
ma1=m1;
ma2=m2;
}
void putmarks()
{
System.out.println("marks obtained");
System.out.println("mark1 :"+ma1);
System.out.println("mark2 :"+ma2);
}
}

interface clas
{
String cla="iii cs b";
void putclas();
int ma3=100;
```

```
}

class result extends mark implements clas
{
int total;
public void putclas()
{
System.out.println("class :"+cla);
}
void display()
{
total=ma1+ma2+ma3;
putnumber();
putmarks();
System.out.println(“mark3: “+ma3);
putclas();
System.out.println("total mark:"+total);
}}

class inter1
{
public static void main (String args[])
{
result ob=new result();
ob.getnumber(100);
ob.getmarks(45,66);
ob.display();
}
}
```

Packages: Putting classes together

We have repeatedly stated that one of the main features of OOP is its ability to reuse the code already created. One way achieving this is by extending the classes and implementing the interfaces we had created as discussed. This limited to reusing the classes within a program. What if we need to use classes from other programs without physically copying them into the program under development?. This can be accomplished in java by using what is known as ***package,*** a concept similar to " class libraries" in other languages. Another way of achieving the reusability in java, therefore, is to use package.

Packages are java's way of grouping a variety of class and/or interfaces together. The grouping is usually done according to functionality. In fact, packages act as "containers" for classes. By organizing our classes into packages we achieve the following benefits:

1. The classes contained in the packages of other programs can be easily reused.
2. In packages, classes can be unique compared with classes in other packages. That is, two classes in two different packages can have the same name. They may be referred by their fully qualified name, comprising the package name and the class name.
3. Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.
4. Packages also provide a way for separating "design" from "coding". First we can design classes and decide their relationships, and then we can implement the java code needed for the methods. It is possible to change the implementation of any method without affecting the rest of the design.

Java packages are therefore classified into two types. The first category is known as ***Java API packages*** and the second is known as ***user defined packages.***

JAVA API PACKAGES:

Java API provides a large number of classes grouped into different packages according to functionality. Most of the time we use the packages available with the java API. Java has a six packages.

Java.lang :

Language support classes. These are classes that java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math function, threads and exception.

Java.util :

Language utility classes such as vectors, hash tables, random numbers, date, etc.

Java.io :

Input/output support classes. They provide facilities for the input and output of data.

Java.awt :

Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

Java.net :

Classes for networking. They include classes for communicating with local computers as well as with internet servers.

Java.applet :

Classes for creating and implementing applets.

USER DEFINED PACKAGES:

User create a packages and process is called user defined packages. Let us see how to create our own packages. We must first declare the name of the package using the **package** keyword followed by a package name. This must be the first statement in a java source file (except for comment and white spaces). Then we define a class, just as we normally define a class. Here is an example

```
package firstpackage;

public class firstclass
{
        ----------------
        ---------------- (body of class)
}
```

Here the package name is **firstpackage**. The class **firstclass** is now considered a part of this package. This listing would be saved as a file called **firstclass.java ,** and located in a directory named **firstpackage.** When the source file is compiled, java will create a **.class** file and store it in the same directory.

Remember that the **.class** files must be located in a directory that has the same name as the package, and this directory should be a subdirectory of the directory where classes that will import the package are located.

To recap, creating our own package involves the following steps:

1. Declare the package at the beginning of a file using the form **package** packagename;
2. Define the class that is to be put in the package and declare it **public.**
3. Create a subdirectory under the directory where the main source files are stored.
4. Store the listing as the classname.java file in the subdirectory created.
5. Compile the file. This creates **.class** file in the subdirectory.

Remember that case is significant and therefore the subdirectory name must match the package name exactly. As pointed out earlier, java also supports the concept of package hierarchy. This is done by specifying multiple names in a package statement, separated by dots.

ACCESSING A PACKAGE:

It will recalled that we have discussed earlier that a java system package can be accessed either using a fully qualified class name or using a shortcut approach through the **import** statement. We use the **import** statement when there are many references to a particular package or the package name is too long and unwieldy.

The same approaches can be used to access the user-defined packages as well. The **import** statement can be used to search a list of packages for a particular class. The general form of **import** statement for searching a class is as follows:

Import package1 [.package2] [.package3].classname;

Here **package1** is the name of the top level package, **package2** is the name of the package that is inside the **package1,** and so on. We can have any number of packages in a package hierarchy. Finally, the explicit **classname** is specified. Note that the statement must end with a semicolon(;). The **import** statement should appear before any class definitions in a source file.

USING A PACKAGE :

Let us now consider some simple programs that will use classes from other packages. The listing below shows a package named **visa** containing a single class **gisa**.

```
package visa;

public class gisa
{
public void show()
{
System.out.println("welcome to all by arun saran");
System.out.println("welcome to package program");
}
}
```

This source file should be named **gisa.java** and stored in the subdirectory **visa** as started earlier. Now compile this java file. The resultant **gisa.class** will be stored in the same subdirectory. Now consider the listing shown below:

```
import visa.gisa;
```

```
public class simi
{
public static void main(String args[])
{
gisa ob=new gisa();
ob.show();
}}
```

This listing shows a simple program that imports the class **gisa** from the package **visa.** The source file should be saved as **simi**.**java** and then compiled. The source file and the compiled file would be saved in the directory of which **visa** was a subdirectory. Now we can run the program and obtain the results.

During the compilation of **simi.java** the compiler checks for the file **gisa.class** in the **visa** directory for information it needs, but it does not actually include the code from **gisa.class** in the file **simi.class.** when the **simi** program is run, java looks for the file **simi**.**class** and loads it using something called *class loader*. Now the interpreter knows that it also needs the code in the file **gisa.class** and loads it as well.

This program compiled and run to obtain the results. The output will be as under

Welcome all by arun saran

Welcome to package program

MULTIPLE PACKAGES:

More than one package is to be created and imported and process is known as multiple packages.

Example:

(program 1)

```
package a1;
public class one
{
int a=10,b=5;
public void show()
{
int c=a+b;
System.out.println("Add value is"+c);
}

public void show1()
{
int c=a*b;
System.out.println("Mul value is"+c);
}
}
```

first create one folder in c: that is (c:\arun) and save this file as one.java.

(program 2)

```
package a1;

public class two
{
int a=10,b=2;
public void cal()
{
int c=a-b;
System.out.println("Sub value is "+c);
}
public void cal1()
{
int c=a/b;
System.out.println("Divide value is "+c);
System.out.println(“End of package program”);
}}
```

this program save as this directory and name is (two.java).

(program 3)

```
import a1.one;
import a1.two;
public class three
{
public static void main(String args[])
{
one ob=new one();
two ob1=new two();
ob.show();
ob.show1();
ob1.cal();
ob1.cal1();
}}
```

This program save as the same directory and save as file name as(three.java).

Save these three program and then compile the first two programs before set path as below.Suppose your java software as place in your system on c: then follow the procedure as follow.

C:\arun> path = c:\jdk1.3\bin> then press enter

C:\arun> javac one.java ,, ,,

C:\arun> javac two.java ,, ,,

This time java to create automatically two files that is **one.class , two.class .**
Then to create subdirectory on package name (that is a1)
C:\arun> md a1 then press enter then copy this two class file on the subdirectory area.
C:\arun>**copy one.class c:\arun\a1** then press enter
C:\arun>**copy two.class c:\arun\a1** then press enter
Then set class path as follows

C:\arun> set classpath = c:\arun;%classpath%;

Then compile the third program
C:\arun> javac three.java then press enter
C:\arun> java three ,, ,, ,,

Then output as follows

```
Add value is 15
Mull value is 50
Sub value is 8
Divide value is 5
End of package program
```

EXCEPTION HANDLING:

Introduction:

Rarely does a program run successfully at its very first attempt. It is common to make mistakes while developing as well as typing a program. A mistake might lead to an error causing the program to produce unexpected results. Errors are the wrongs that can make a program go wrong.

An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not terminate or crash during execution.

Types of errors:

Errors may broadly be classified into two categories:

Compile-time errors

Run-time errors.

Compile-time errors:

All syntax errors will be detected and displayed by the java compiler and therefore these errors are known as compile-time errors. Whenever the compiler displays an error, it will not create the **.class** file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

Example:

```
/* this program contains an error */
        Class a
        {
                public static void main(String args[])
                {
                        System.out.pritnln(“hello soft”)     //missing
                }
        }
```

The java compiler does a nice job of telling us where the errors are in the program. For example, if we have missed the semicolon at the end of the print statement in this program.

The following message will be displayed in the screen.

```
a.java :7: ';' expected
System.out.println("hello soft")
^          1 error
```

we can now go to the appropriate line, correct the error, and recompile the program. Most of the compile-time error are due to typing mistakes. Typographical errors are hard to find. We may have to check the code word by word, or even character by character.

The most common problems are:

* Missing semicolons
* Missing (or mismatch of) brackets in classes and methods.
* Misspelling of identifiers and keywords
* Missing double quotes in strings
* Use of undeclared variables
- Use of = in place of = = operator

Other errors we may encounter are related to directory paths. An error such as

```
Javac : command not found
```

Means that we have not set the path correctly. We must ensure that the path includes the directory where the java executables are stored.

Run – time errors:

Sometimes, a program may compile successfully creating the **.class** file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are

- Dividing an integer by zero
- Accessing an element that is out of the bounds of an array
- Trying to store a value into an array of an incompatible class or type

- Trying to cast an instance of a class to one of its subclass
- Attempting to use a negative size for an array
- Converting invalid string to a number
- Accessing a character that is out of bounds of a string

When such errors are encountered, java typically generates an error message and aborts the program.

Exception:

An exception is a condition that is caused by a run-time error in the program. When the java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it (i.e., informs us that an error has occurred).

If the exception object is not caught and handled properly, the interpreter will display an error message and will terminate the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as ***exception handling.***

The purpose of exception handling mechanism is to provide a means to detect and report an " exceptional circumstance" so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following tasks:

1. Find the problem (**Hit** the exception)
2. Inform that an error has occurred (**throw** the exception)
3. Receive the error information (**catch** the exception)
4. Take corrective actions (**Handle** the exception)

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions.

When writing programs, we must always be on the lookout for places in the program where an exception could be generated. Some common exception that we must watch out for catching are listed in the table

Common java exception

Exception type	cause of exception
ArithmeticExceptin	Arithmetic errors such as division by zero
ArrayIndexOutOfBoundsException	Array index out of bounds
ArrayStoreException	Assignment to an array element of an incompatible type
ClassCastException	Invalid cast
NegativeArraySizeException	Array created with a negative size.
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failures, such as inability to read from a file
NullPointerException	Invalid use of null reference
NumberFormatExceptoin	Invalid conversion of a string to a numeric format
InterruptedException	One thread has been interrupted by another thread
ClassNotFoundException	class not found

Java exception handling is managed via five keywords :

try, catch, throw, throws , finally

Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch)** and handle it in some rational manner. System-generated exception are automatically thrown by the java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed before a method returns is put in a **finally** block.

Syntax of try – catch

```
try
{
      statement;
}
catch (Exception-type e)
{
      statement;
 }
```

Example :

```
class exe
{
 public static void main (String args[])
 {
 try
  {
     int a = 10,b=0;
     int c = a /b;
     System.out.println("divide value is "+c);
     }
     catch(ArithmeticException e)
     {
     System.out.println("Divide by zero");
     }
     }}
```

This program to run then the output as follows

Divide by zero

Note that the program did not stop at the point of exceptional condition. It catches the error condition, prints the error message, and then continues the execution.

Multiple catch statements:

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statements is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block.

The following example traps two different exception types

```
class exe1
{
public static void main(String args[])
{

try
{
int a=2;

System.out.println(" a value is "+a);

int b=40/a;

int c[]={ 1};
c[40]=90;

}

catch(ArithmeticException e)
{

System.out.println("divide by 0 :"+e);

}

catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("array index obj: ");
}

System.out.println("after try / catch blocks");

}
}
```

Nested try statements:

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try.** Each time a **try** statement is entered, the context of that exception is pushed on the stack. In an inner statement is entered, the context of that exception is pushed on the stack. In an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the java run-time system will handle the exception. Here is an example that uses nested **try** statements.

```
class exe2
{
public static void main(String args[])
{
try
{

int a=10;
int c,b;
b=Integer.parseInt(args[0]);
c=a/b;
System.out.println(c);
try
{
int d,e,f;
d=Integer.parseInt(args[1]);
e=Integer.parseInt(args[2]);
f=d*e;
System.out.println("mul value is "+f);
}
catch(NumberFormatException e)
{
System.out.println("give only integer value");
}
}
catch(ArithmeticException e)
{
System.out.println("divide by zero ");
}
}
}
```

Finally

java supports another statement known as **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements. f**inally** block can be used to handle any exception generated within a try block. It may be added immediately after the try block or after the last catch block shown as follows

```
try
{
        statement;
}
finally
{
        statement;
}
```

example:

```
class exe3
{
public static void main(String args[])
{

int a,b,c;
try
{

a=Integer.parseInt(args[0]);
b=Integer.parseInt(args[1]);
c=a/b;
System.out.println("division value is"+c);
}

finally
{
System.out.println("final block");
a=10;b=20;

c=a+b;
System.out.println("adition value is"+c);
}
System.out.println("simi");
}}
```

throw

so far you have only been catching exceptions that are thrown by the java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here

throw *throwableinstance;*

here, throwableinstance must be an object of type **throwable** or a subclass of **throwable**. Simple types , such as **int** or **char,** as well as non-**throwable** classes, such as **string** and **object**, cannot be used as exceptions. There are two ways you can objatin a **thrwoable** object: using a parameter into a **catch** clause, or creating one with the **new** operator.

The flow of execution stops immediately after the **throw** statement: any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statements is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
class exe112
{
public static void main(String args[])
{
try
{
int a,b,c;
a=Integer.parseInt(args[0]);
b=Integer.parseInt(args[1]);
c=a/b;
System.out.println("divide value is"+c);
}
catch(ArithmeticException e)
{
System.out.println("main block the error is ");
throw e;
}
catch(NumberFormatException e)
{
System.out.println("second type of error");
throw e;
}
}}
```

Throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **error** or **RuntimeException** , or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a **throws** clause:

```
Type method-name(parameter-list) throws exception-list
{
body of method;
}
```

Example:

```
class my extends Exception
{
 int x;
my(int a)
{
x = a;
}
public String toString()
{
return "My exception ("+ x +") ";
}
}
class exe12
{
public static void compute(int a) throws my
{
System.out.println("given value(" + a +")");
if(a  >10)

throw new my(a);

System.out.println("Normal exit");
}
```

```
public static void main(String args[])
{
try
{
compute(1);
compute(20);
}
catch(my e)
{
System.out.println("Exception exit  "+e);
}
}
}
```

The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **throwable.** Thus, all exceptions, including those that your create, have the methods defined by **throwable** available to them. They are shown in table you may also wish to override one or more of these methods in exception classes that you create.

Method	Description
String getMessage	Returns a description of the exception
String toString()	Returns a **String** object containing a description of the exception. This method is called by **println()** when outputting a **throwable** object.
void printStackTrace (PrintStream stream)	Sends the stack trace to the specified stream.
void printStackTrace (PrintWriter stream)	Sends the stack trace to the specified stream.

MULTITHREADING

Those who are familiar with the modern operating systems such as Windows 95 may recognize that they can execute several programs simultaneously. This ability is known as ***multitasking***. In system's terminology, it is called ***multithreading.***

Multithreading is a conceptual programming paradigm where a program(process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. Each part of subprograms is called ***thread.*** For example, one subprogram can display an animation on the screen while another may build the next animation to be displayed.

The Main Thread:

When a java program starts up, one thread begins running immediately. This is usually called the ***main thread*** of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons

- It is the thread from which other " child" threads will be spawned.
- Often it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( ),** which is a **public static** member of **Thread**. Its general form is shown here,

static Thread currentThread( )

Example:

```
class th1
{
public static void main(String args[])
{
Thread t1=Thread.currentThread();
System.out.println("current thread :"+t1);

t1.setName("My Thread  ");
System.out.println("After name change" +t1);
try
```

```
{
for(int i=1;i<=5;i++)
{
System.out.println(i);
Thread.sleep(1000);
}}

catch(InterruptedException e)
{
System.out.println("main thread interrupted");
}}
}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( )**, and this reference is stored in the local variable t1. Next, the program displays information about the thread. The program then calls **setName( )** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts 1 to 5 pausing one second between each line. The pause is accomplished by the **sleep( )** method. The argument to **sleep( )** specifies the delay period in milliseconds.

Notice the **try / catch** block around this loop. The **sleep( )** method in **Thread** might throw an **InterruptedException.** This would happen if some other thread wanted in interrupted this sleeping one. This example just prints a message if it gets interrupted. Here is the output generated by this program:

```
Current thread: Thread[main, 5, main]
After name change: Thread[My Thread,5,main]
1
2
3
4
5
```

The Thread class and the Runnable Interface:

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable.** The **Thread** class defines several methods that help manage threads.

getName:
It obtain a thread's Name.

getPriority:
It obtain a thread's priority.

isAlive:
Determine if a thread is still running.

join:
Wait for a thread to terminate.

run:
Entry point for the thread.

sleep:
suspend a thread for a period of time.

start:
start a thread by calling its run method.

Life cycle of Thread:

Running:
Running state used for thread hold on particular CPU.

Ready to run:
At any time CPU free then that time get ready into thread for CPU.

Suspended:
Running thread suspended into CPU

Blocked:

A thread can be blocked when waiting for the other resources in a CPU. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

Terminated:

At any time any thread can be terminated, thread can not be resources or re start.

Resumed:

Resumed used for continues thread in a particular CPU.

Creating a Thread:

In the most general sense, you create a thread by instantiating an object of type **Thread.** Java defines two ways in which this can be accomplished.

Implement the **Runnable** interface.

Extend the **Thread** class, itself.

Implementing Runnable:

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable.** To implement **Runnable**, a class need only implement a single method called **run( )** , which is declared like this:

Public void run( )

Inside **run( )** , you will define the code that constitutes the new thread. It is important to understand the **run( )** can call other methods, use other classes, and declare variables, just like the main thread .

Syntax:

```
Class x implements Runnable
{
        ……
        ……
}
```

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread.** In essence, **start( )** executes a call to **run( ).** The **start( )** method is shown here:

void start( )

Example :

```
class exthread implements Runnable
{
exthread()
{
Thread t=new Thread(this,"demo thread");
System.out.println("Child thread"+t);
t.start();
}
public void run()
{
try
{
for(int i=1;i<5;i++)
{
System.out.println("child thread i value is "+i);
Thread.sleep(500);
}}
catch(InterruptedException e)
{
System.out.println("child interrupted");
}
System.out.println("exiting the child thread");
}}
class th4
{
public static void main(String args[])
{
new exthread();
try
{
for(int i=10;i<15;i++)
{
System.out.println("main thread "+i);
Thread.sleep(1000);
}}
catch(InterruptedException e)
{
System.out.println("Main thread interrupted");
}
System.out.println("exiting the main thread");
}}
```

Extending the thread class:

The second way to create a thread is to create a new class the extends **Thread,** and then to create an instance of that class. It includes the following steps:

Declare the class as extending the **Thread** class.

Implement the **run( )** method that is responsible for executing the sequence of code that the thread will execute.

Create a thread object and call the **start( )** method to initiate the thread execution.

Declaring the Class:

The **Thread** class can be extended as follows:

```
Class MyThread extends Thread
{
        ……
        ……
}
```

Now we have a new type of thread **MyThread**.

Implementing the run( ) Method:

The **run( )** method has been inherited by the class **MyThread.** We have to override this method in order to implement the code to be executed by our thread. The basic implementation of **run( )** will look like this:

```
Public void run( )
{
        …….
        ……..
}
```

When we start the new thread, java calls the thread's **run( )** method, so it is the **run( )** where all the action takes place.

Starting New Thread:

To actually create and run an instance of our thread class, we must write the following:

```
MyThread aThread = new MyThread( );
AThread.start( );
```

The first line instantiates a new object of class **MyThread**. Note that this statement just creates the object. The thread that will run this object is not yet running. The thread is in a newborn state. *Arun*

The second line calls the **start( )** method causing the thread to move into the *runnable* state. Then, the java runtime will schedule the thread to run by invoking its **run( )** method. Now, the thread is said to be in the *running* state.

Example:

```
class a extends Thread
{
public void run()
{
for(int i=1;i<=5;i++)
{
System.out.println("\t from thread a :i = "+i);
}
System.out.println("exit form a");
}}

class b extends Thread
{
public void run()
{
for(int j=10;j<=15;j++)
{
System.out.println("\t from thread b :j = "+j);
}
System.out.println("exit from b");
}}

class c extends Thread
{
public void run()
{
```

```
for(int k=15;k<=20;k++)
{
System.out.println("\t from thread c :k = "+k);
}
System.out.println("Exit from c");
}}

class th3
{
public static void main(String args[])
{
new a().start();
new b().start();
new c().start();
}}
```

THREAD PRIORITY:

In java, each thread is assigned apriority, which affects the order in which it is scheduled for running. The threads that we have discussed so far are of the same priority. The threads of the same priority are given equal treatment by the java scheduler and, therefore, they share the processor on first-serve basis.

Syntax of thread priority

ThreadName.setPriority(intNumber);

The **intNumber** is an integer value to which the thread's priority is set. The **Thread** class defines several priority constants:

MIN_PRIORITY = 1

NORM_PRIORITY= 5

MAX_PRIORITY = 10

The **intNumber** may assume one of these constants or any value between 1 and 10. Note that the default setting is **NORM_PRIORITY.**

Most user-level processes should use **NORM_PRIORITY,** plus or minus. Back-ground tasks such as network I/O and screen repainting should use a value very near to the lower limit. We should be very cautious when trying to use very high priority values.

Example:

```
class a extends Thread
{
public void run()
{
System.out.println("thread a started");

for(int i=1;i<=4;i++)
{
System.out.println("\t from thread a : i="+i);
}
System.out.println("exit from a");
}}


class b extends Thread
{
public void run()
{
System.out.println("thread b started");
for(int j=10;j<=14;j++)
{
System.out.println("\t from thread b : j="+j);
}
System.out.println("exit from b");
}}

class c extends Thread
{
public void run()
{
System.out.println("thread c started");
for(int k=20;k<25;k++)
{
System.out.println("\t from thread c : k= "+k);
}
System.out.println("exit from c");
}}

class th13
{
public static void main(String args[])
{
a ob = new a();
```

```
b ob1 = new b();
c ob2 = new c();

ob.setPriority(Thread.MIN_PRIORITY);
System.out.println("start thread a");
ob.start();
System.out.println("start thread b");
ob1.start();
System.out.println("start thread c");
ob2.start();
System.out.println("end of main thread");
```

In this program class b and class c will be work on same time but the class a will be work on after working on b & c class. That is class a will be work on last because it will be set on lower priority.

Synchronization

When two or more threads need access to a shared resource, 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 achieved is called *synchronization.*

Example:

```
class a extends Thread
{
public void run()
{
try
{
sync.display();
Thread.sleep(250);
}
catch(InterruptedException e)
{
System.out.println("Interrupted");
}
System.out.println("exit form a");
}}
```

```
class sync
{
public static synchronized void display()
{
for(int x=200;x<203;x++)
{
System.out.println("a thread x value is"+x);
}}}
class b extends Thread
{
public void run()
{
try
{
for(int j=1;j<5;j++)
{
System.out.println("b thread j value is "+j);
Thread.sleep(500);
}}

catch(InterruptedException e)
{
System.out.println("interrupted");
}
}}

class th12
{
public static void main(String args[])
{
new a().start();
new b().start();
}}
```

In this program first class a process completed then the class b process will be started.

Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is ***deadlock*** , which occurs when two threads have circular dependency on a pair of synchronized objects.

For example, suppose one thread enters the monitor on object x and another thread enters the monitor on object y. if the thread in x tries to call any synchronized method on y, it will block as expected. However, if the thread in y, in turn, tries to call any synchronized method on x, the thread waits forever, because to complete.

Deadlock is a difficult error to debug for two reasons.

In general, it occurs only rarely, when the two threads time-slice in just the right way.

It may involve more than two threads and two synchronized objects.(that is , deadlock can occur through a more convoluted sequence of events than just described).

```
class firstt
{
synchronized void display(second s)
{
System.out.println("arun");
try
{
Thread.sleep(1000);
}
catch(Exception e)
{
System.out.println("one interrupted");
}
System.out.println("kumar");
s.display1();
//here deadlock occur
}
synchronized void display1()
{
System.out.println("anbu");
}}

class second
{
synchronized void show(firstt f)
```

```
{
System.out.println("simi");
try
{
Thread.sleep(1000);
}
catch(Exception e)
{
System.out.println("first interrupted");
}
System.out.println("ammu");
f.display1();
// here dead lock occur
}
synchronized void display1()
{
System.out.println("arasan");
}}

class deadlo2 extends Thread
{
firstt f=new firstt();
second s=new second();

deadlo2()
{
start();
f.display(s);
System.out.println("back in main thread");
}
public void run()
{
s.show(f);
System.out.println("back in other thread");
}
public static void main(String args[])
{
new deadlo();
}}
```

output of this program is arun
simi
kumar
ammu

then deadlock entered then we have to press ctrl + c then control come back dos prompt.

SUSPEND, RESUME A THREAD:

```
//example program for suspend, resuming thread

class ones extends Thread
{
boolean suspendFlag;
String name;
ones(String tname)
{
name=tname;
suspendFlag=false;
start();
}

public void run()
{
try
{
for(int i=15;i>0;i--)
{
System.out.println(name +" : "+ i);
Thread.sleep(200);

synchronized(this)
{
while(suspendFlag)
{
wait();
}}

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

void mysuspend()
{
suspendFlag=true;
}

synchronized void myresume()
{
```

```
suspendFlag=false;
notify();       // it is used to wake up the thread
}
}

class th161
{
public static void main(String args[])
{
ones o=new ones("arun");
ones o1=new ones("kumar");

try
{
Thread.sleep(1000);

o.mysuspend();
System.out.println("suspending thread one");
Thread.sleep(1000);

o.myresume();
System.out.println("resuming thread one");


o1.mysuspend();
System.out.println("suspendint thread two");
Thread.sleep(1000);
o1.myresume();
System.out.println("resuming thread two");
}
catch(Exception e)
{
System.out.println("interrupted");
}
}
}
```

Stream I/O and Files

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

Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket.

Likewise, an output stream may refer to the console, a disk file, or a network connection.

Types of Stream:

Java defines two types of Streams:

1. Byte Streams 2. Character Streams.

Byte Stream:

Byte streams provide a convenient means fro handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. Byte streams are defined by using two classes hierarchies.

At the top are two abstract classes : **InputStream** and **OutputStream.** Each of these abstract classes has several concrete subclasses, that handle the differences between various devices, such as disk files, network connections and even memory buffers.

The byte stream classes are as bellow:

Stream Classes	**Meaning**
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
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
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
PrintStream	Output stream that contains **print( )** &**println( )**
RandomAccessFile	supports random access File I/O

The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most implement are **read( )** and **write( )**, which respectively, read and write bytes of data. Both methods are declared as abstract inside **InputStream & OutputStream.**

Character Stream:

Character streams are defined by using two class hierarchies. At the top are two abstract classes, **Reader & Writer.** These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes are below:

Stream Classes	**Meaning**
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
InputStreamReader	Input stream that translates bytes to characters
PrintWriter	Output stream that contains **print( ) & println( )**
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream Output.

Reading console Input:

In java, console input is accomplished by reading from **System.in.** to obtain a character-based stream that is attached to the console, you wrap **System.in** in a **BufferedReader** object, to create character stream. **BufferedReader** supports a buffered input stream. It is most commonly used constructor is shown here:

BufferedReader(Reader *inputReader)*

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. Reader is abstract class. One of its concrete subclasses is **InputStreamReader,** which converts bytes to charcters. To obtain an **InputStreamReader** object that is linked to **System.in,** use the following constructor

InputStreamReader(InputStream *inputStream)*

Because **System.in** refers to an object of type **InputStream,** it can be used for *inputStream.* Putting it all together, the following line of code creates **BufferedReader** that is connected to the keyword:

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

After this statement executes, **br** is a character based stream that is linked to the console through **System.in.**

Example: (String Functions)

```
import java.io.*;
class str
{
public static void main (String args[])throws IOException
{
InputStreamReader br=new InputStreamReader(System.in);
BufferedReader r=new BufferedReader(br);
String name,a,b,c;
System.out.println("Enter the first string");
name=r.readLine();

System.out.println("Enter the string");
a=r.readLine();

System.out.println("given name is "+ name);
System.out.println("given name is "+ a);
```

```
System.out.println("comp"+name.compareTo(a));

int l;
l=name.length();
System.out.println("given name length is "+l);
System.out.println("   ");
System.out.println("given name length is "+ a.length());
System.out.println("   ");

System.out.println("concat string is "+ name.concat(a));
System.out.println("   ");

System.out.println("Equal string " +name.equals(a));
System.out.println("   ");

System.out.println("Upper case letter of the given name"+name.toUpperCase());
System.out.println("   ");

System.out.println("Upper case letter of the given name"+a.toLowerCase());
System.out.println("   ");

StringBuffer x=new StringBuffer(a);
x.reverse();

String y=new String(x);
System.out.println("Reverse of the string is: "+y);
  if(a.compareTo(y)==0)
  System.out.println("Given String is polyndrome");
  else
  System.out.println("not polyndrome strig is ");

}}
```

PrintWriter Class:

Although using **System.out.** to write to the console is still permissible under java its use is recommended mostly for debugging purposes. For real-world programs, the recommended method of writing to the console when using java is through a **PrintWriter** stream. **PrintWriter** is one of the character-based classes. Using a character based class for console output makes it easier to internationalize your program.

PrintWriter defines several constructors. The one we will use is shown here:

PrintWriter(OutputStream *outputStream*, Boolean *flushOnNewline*)

Here , *outputStream* is an Object of type **OutputStream,** and *flushOnNewline* controls whether java flushes the output stream ever time a **println( )** method is called. If *flushOnNewline* is **true,** flushing automatically takes place. If **false** , flushing in not automatic.

PrintWriter pw=new PrintWriter(System.out, true);

Example:

```
import java.io.*;
class pwriter
{
public static void main(String args[])
{
PrintWriter pw=new PrintWriter(System.out,true);
pw.println("this is a string");
int i=9;
pw.println(i);
}}
```

FILES:

We have used variables and arrays for storing data inside the programs. This approach poses the following problems.

The data is lost either when a variable goes out of scope or when the program is terminated. That is , the storage is temporary.

It is difficult to handle large volumes of data using variables and arrays.

We can overcome these problems by storing data on *secondary storage devices* such as floppy disks or hard disks. The data is stored in these devices using the concept of *files.*

A file is a collection of related records placed in particular area on the disk. Storing and managing data using files is known as *file processing* which includes tasks such as creating files, updating files and manipulation of data.

USING THE FILE CLASS:

The **java.io,** package includes a class known as the **file** class that provides support for creating files and directories. The class includes several constructors for instantiating the **File** objects. This class also contains several methods for supporting the operations such as

Creating a File , Opening a File, Closing a File, Deleting a File, Getting the name of the File, Renaming a File, checking whether the file is readable, checking whether the file is writable.

INPUT / OUTPUT EXCEPTIONS:

When creating files and performing i/o operations on them, the system may generate i/o related exceptions. The basic i/o exception classes and their functions are given below

I/O Exception class	**Function**
EOFException	Signals that an end of the file or end of stream has been reached unexpectedly during input
FileNotFoundException	Informs that a file could not found
InterruptedIOException	Warns that an I/O operations has be interrupted
IOException	signals that an I/O exception of some sort has occurred.

CREATION OF FILES:

If we want to create and use a disk file, we need to decide the following about the Intended purpose.

Suitable name of the file
Data type to be stored
Purpose (reading, writing, or updating)
Method of creating the file.

READING / WRITING CHARACTES:

As pointed out earlier, subclasses of **Reader & Writer** implement streams that can handle characters. The two subclasses used for handlingcharacters in files are **FileReader** (for reading characters) and **FileWriter(**for writing characters).

In this program , two file stream classes to copy the contents of a file amed input.dat" into a file called "output.dat".

```
//coping charcters from one file into another
import java.io.*;
class file1
{
public static void main(String args[])
{
File infile=new File("input.dat");
File outfile=new File("output.dat");
FileReader ins=null;
FileWriter outs=null;
try
{
ins=new FileReader(infile);
outs=new FileWriter(outfile);
int ch;
while((ch=ins.read()) !=-1)
{
outs.write(ch);
}
}
catch(IOException e)
{
System.out.println(e);
}
finally
{
```

```
try
{
ins.close();
outs.close();
}
catch(IOException e)
{
System.out.println("interupted");
}}}}
```

this program is very simple. It creates two file objects **inFile & outFile** and initializes them with “input.dat”, and “output.dat” respectively using the following code.

```
File infile= new File(“input.dat”);
File outfile=new File(“output.dat”);
```

The program then creates two file stream objects **ins & outs** and initializes them with “null” as follows.

```
FileReader ins=null;
FileWriter outs=null;
```

These Streams are then connected to the named files using the following codes

```
ins=new FileReader(infile);
outs=new FileWriter(outfile);
```

This connects **infile** to the **FileReader** stream **ins** and **outfile** to the **FileWriter** stream outs. This essentially means that the files “input.dat” and “output.dat” are opened. The statements ch=ins.read( )

Reads a character from the **infile** through the input stream **ins** and assigns it to the variable ch similarly, the statement

```
outs.write(ch);
```

writes the character stored in the variable **ch** to the **outfile** through the output stream outs. The character-1 indicates the end of the file and therefore the code

```
while((ch=ins.read( )) != -1)
```

causes the termination of the while loop when the end of the file is reached. The statements

```
ins.close( );  outs.close( );
```

enclosed in the **finally( )** clause close the files created for reading and writing. When the program catches an I/O exception, it prints a message and then exits from execution.

Example:

FileInputStream, FileOutputStream:

```
import java.io.*;
class file3
{
public static void main(String args[])
{
try
{
FileOutputStream fos=new FileOutputStream("arun.txt");
for(int i=1;i<10;i++)
{
fos.write(i);
}
fos.close();
}
catch(IOException e)
{
System.out.println(e);
}}}
```

In this program create a new file that is “arun.txt” and write the numbers between 1 and 10 and close that file.

```
import java.io.*;
class file31
{
public static void main(String args[])
{
try
{
FileInputStream fis=new FileInputStream("arun.txt");
int i;
while((i=fis.read())!=-1)
{
System.out.println(i);
}
fis.close();
}
catch(IOException e)
{
System.out.println(e);
}}}
```

In this program to open a new file that is “arun.txt” and read the message and print the message to your console normal screen.

SequenceInputStream:

The **SequenceInputStream** class allows you to concatenate multiple I**nputStream.** The construction of **SequenceInputStream** is different from any other **InputStream. A SequenceInputStream** constructor use either a pair of **InputStream** or an **Enumeration** of **InputStream** as its argument.

SequenceInputStream(InputStream first, InputStream second)
SequenceInputStream(Enumeration *stream* Enum)

```
import java.io.*;
class file5
{
public static void main(String args[])throws IOException
{
FileInputStream file1=null;
FileInputStream file2=null;

SequenceInputStream file3=null;

file1=new FileInputStream("one.txt");
file2=new FileInputStream("two.txt");
file3=new SequenceInputStream(file1,file2);


BufferedInputStream inb=new BufferedInputStream(file3);
BufferedOutputStream ob=new BufferedOutputStream(System.out);

int ch;
while((ch=inb.read())!=-1)
{
ob.write((char)ch);
}
inb.close();
ob.close();
file1.close();
file2.close();
}}
```

In this program to create two file that is “one.txt” and “two.txt” and write some message. Then to store file3 that is SequenceInputStream it includes one.txt , two.txt (that is file1,file2). Then file3 to be stored on BufferedInputStream. Then use of BufferedOutputStream to read the character from file3 and print the message on to the screen.

DataInpurStream , DataOutputStream:

The **DataInputStream** class extends **FilterInputStream** and implements DataInputStream. This class provides this constructor:

DataInputStream(InputSteam is)

Here, is the input stream.

The **DataInput** interface defines methods that can be used to read the simple java types from a byte input stream.

Example:

```
import java.io.*;
class file7
{
public static void main(String args[])
{
try
{
FileOutputStream fos=new FileOutputStream("kumar.txt");

DataOutputStream dos=new DataOutputStream(fos);

dos.writeBoolean(false);
dos.writeByte(Byte.MAX_VALUE);
dos.writeChar('A');
dos.writeDouble(Double.MAX_VALUE);
dos.writeFloat(Float.MAX_VALUE);
dos.writeInt(Integer.MAX_VALUE);
dos.writeLong(Long.MAX_VALUE);
dos.writeShort(Short.MAX_VALUE);

fos.close();
}
catch(Exception e)
{
System.out.println("Exception :"+e);
}
}}
```

In this program to create “kumar.txt” and write the message the max value of int,float,long,short data type values.

```
import java.io.*;
class file71
{
public static void main(String args[])
{
try
{
FileInputStream fis=new FileInputStream("kumar.txt");

DataInputStream dis=new DataInputStream(fis);

System.out.println(dis.readBoolean());
System.out.println(dis.readByte());
System.out.println(dis.readChar());
System.out.println(dis.readDouble());
System.out.println(dis.readFloat());
System.out.println(dis.readInt());
System.out.println(dis.readLong());
System.out.println(dis.readShort());
dis.close();
}
catch(Exception e)
{
System.out.println("Exception :"+e);
}
}}
```

In this program to open the file “kumar.txt” and read the message and print the message on the screen.

RANDOM ACCESS FILES:

The stream classes examined in the previous sections can only use sequential access to read and write data in File. The **RandomAccessFile** class allows you to write programs that can seek to any location in a file and read or write data at the point. It also supports positioning requests- that is, you can position the *file pointer* within the file. It has two constructors:

RandomAccessFile(File *fileobj*, String *access*) throws FileNotFoundException

RandomAccessFile(String filename, String access) throws FileNotFoundException

In the first form, *fileobj* specifies the name of the file to open as a **File** object. In the second form, the name of the file is passed in *filename*. In both cases, *access* determines what type of file access is permitted. If it is "r" , then the file can be read, but not written. If it is "rw" , then the file is opened in read-write mode.

The method **seek( )** shown here, is used to set the current position of the file pointer within the file:

Void seek(ling *newpos*) throws IOException

Here , *newpos* specifies the new position, in bytes, of the file pointer from the beginning of the file. After call to **seek( )**, the next read or write operation will occur at the new file position.

Example:

```
import java.io.*;

class file8
{
public static void main(String args[])
{
RandomAccessFile file=null;

try
{
file=new RandomAccessFile("rand.dat","rw");

file.writeChar('x');
file.writeInt(888);
file.writeDouble(89.456);
```

```
file.seek(0);//go to the begining

System.out.println(file.readChar());
System.out.println(file.readInt());
System.out.println(file.readDouble());


file.seek(2);//go to the second item

System.out.println(file.readInt());

file.seek(file.length());//go to the end and append false to the file
file.writeBoolean(false);

file.seek(3);
System.out.println(file.readBoolean());
file.close();
}
catch(IOException e)
{
System.out.println(e);
}
}}
```

In this program first to create a new file that is “rand.dat” that file is read and write file. Then to write three message on that file. Then read on that file and print the message on the screen one by one.

Then use of the **seek( )** method and print the second message. Then use of **length** class to add another message on that file.

Then use of the seek method to print the new (that is fourth) message on the screen. This process is only available on RandomAccessFile method only.

The output as follows:

```
x
888
89.456
888
true
```

AWT Classes:

The AWT contains numerous classes and methods that allow you to create and manage windows. The AWT when creating your own applets or stand-alone programs. The main purpose of the AWT is to support applet windows, it can also be used to create stand-alone windows that run in a GUI environment, such as windows. Most of the examples are contained in applets, so to run them, you need to use an applet viewer or a java-compatible web browser.

The AWT classes are contained in the **java.awt** package. It is one of java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Some AWT classes below:

Class	Description
AWTEvent	Encapsulates AWT events.
BorderLayout	The border layout manager. It use five components: North, South, East, West and Center.
Button	Creates a push button control.
Canvas	A blank, semantics-free window.
Checkbox	Creates a check box control
CheckboxGroup	Creates a group of check box controls.
Choice	creates a pop-up list.
Component	An abstract super class for various AWT components.
Container	A subclass of **Component** that can hold other components.
Dialog	Creates a dialog window.
FlowLayout	the flow layout manager. Flow layout positions components left to right, top to bottom.
Font	Encapsulates a type font.
FontMetrics	Encapsulates various information related to a font. This information helps you display text in a window
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.
Graphics	encapsulates the graphics context. It is used by the various output methods to display output in a window.
GridLayout	the grid layout manager. To displays components in a two-dimensional grid.
Image	Encapsulates graphical images.
Label	Creates a label that displays a string.

Class	Description

List	Creates a list from which the user can choose.
Menu	Creates a pull-down menu.
Panel	the simplest concrete subclass of **Container.**
Rectangle	Encapsulates a rectangle.
Scrollbar	Creates a scroll bar control
Textarea	Creates a multi line edit control.
TextField	Creates a single-line edit control.
Window	Creates a window with no frame, no menu bar, and no title.

WINDOW FUNDAMENTALS:

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from **Panel,** which is used by applets, and those derived from **Frame**, which creates a standard window.

Component:

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component.**

It responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

Container:

The **Container** class is a subclass of **Component.** It has additional methods that allow other **Component** objects to be nested within it. A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers.

Panel:

The **Panel** class is concrete subclass of **Container.** It doesn't add any new methods; it simply implements **Container.** A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the super class for **Applet.** When screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does

not contain a title bar, menu bar, or border. When you run an applet using an applet viewer, the applet viewer provides the title and border.

Other components can be added to a **Panel** object by its **add( )** method. Once these components have been added, you can position and resize them manually using the **setLocation( ), setSize( ), or setBounds( )** methods defined by **Component.**

Window:

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally , you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame,** described next.

Frame:

Frame encapsulates what is commonly thought of as a "window". It is subclass of window and has a title bar, menu bar, borders, and resizing corners.

Canvas:

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas. Canvas** encapsulates a blank window upon which you can draw.

Working with Frame Windows:

After the applet, the type of window you will most often create is derived from **Frame.** You will use it to create child windows with applets, and top-level or child windows for applications. Here are two of **Frame's** constructors:

Frame( )
Frame(String title)

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by title.

Hiding and Showing a Window:

After a frame window has been created, it will not be visible until you call **setVisible( ).** Its signature is shown here:

void setVisible(Boolean *visible flag)*

The component is visible if the argument to this method is **true.** Otherwise, it is hidden.

Setting a Window's Title:

You can change the title in a frame window using **setTitle( ),** which has this general form:

void setTitle(String *newTitle)*

here, *newTitle* is the new title for the window.

Creating a Frame Window in an Applet:

While it is possible to simply create a window by creating an instance of **Frame,** you will seldom do so, because you will not be able to do much with it. Creating a new frame window from within an applet is actually quite easy. First, create a subclass of **Frame**. Next, override any of the standard window methods, such as **init( ), start( ), stop( ), and paint( ).** Finally, implement the **windowClosing( )** method of the **WindowListener** interface, calling **setVisible(false)** when the window closed.

Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

/*
<applet code= "app811" width = 300 height = 50>
</applet>
*/

class o extends Frame
{
public void paint(Graphics g)
{
g.drawString("this is a frame window",50,59);
}}

public class app811 extends Applet
{
Frame f;
public void init()
{
f=new o();
f.setSize(250,250);
f.setVisible(true);
}
public void paint(Graphics g)
```

```
{
g.drawString("this is an applet window",10,20);
showStatus("this is applet window status bar");
}
}
```

Working with Graphics:

The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. The origin of each window is at the top-left corner and is 0,0. coordinates are specified in pixels. All output to a window takes place through a graphics context. A *graphics context* is encapsulated by the **Graphics** class and is obtained in two ways:

It is passes to an applet when one of its various methods, such as **paint( )** or **update( )** is called.

It is returned by the **getGraphics( )** method of **Component.**

The **Graphics** class defines a number of drawing functions. Each shape can be drawn edge-only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default.

Drawing Lines:

Lines are drawn by means of the **drawLine( )**method, shown here:

```
void drawLine(int startx, int starty, int endx, int endy)
```

drawLine( ) displays a line in the current drawing color that begins at startx,starty and ends at endx, endy.

Drawing Rectangle:

The **drawRect( ) & fillRect( )** methods display an outlined and filled rectangle, respectively. They are shown here:

```
void drawRect(int top, int left, int width, int height)
void  fillRect(int top, int left, int width, int height)
```

The upper-left corner of the rectangle is at top,left. The dimensions of the rectangle are specified by width and height.

To draw a rounded rectangle, use **drawRoundRect( ) or fillRoundRect( ),**both shown here

void drawRoundRect(int top, int left, int width, int height, int xDiam, int yDiam)

void fillRoundRect(int top, int left, int width, int height, int xDiam, int yDiam)

A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at *top,left*. The dimensions of the rectangle are specified by width and height. The diameter of the rounding arc along the x axis is specified by xDiam. The diameter of the rounding arc along the y axis is specified by yDiam.

Drawing Ellipses and Circles:

To draw an ellipse, use **drawOval( ).** To fill an ellipse, use **fillObval( ).** These methods as shown here:

void drawOval(int top, int left, int width, int height)

void fillOval(int top, int left, int width, int height)

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by top,left and whose width and height are specified by width and height.

Drawing Arcs:

Arcs can be drawn with **drawArc( ) & fillArc( ),** shown here:

void drawArc(int top, int left, int width, int height, int startAngle,int sweep Angle)

void fillArc(int top, int left, int width, int height, int start Angle, int sweep Angle)

The arc is bounded by the rectangle whose upper-left corner is specified by top, left and whose width and height are specified by width and height. The arc is drawn from start angle through the angular distance specified by sweep angle. Angles are specified in degrees. Zero degree is on the horizontal, at the three o'clock position. The arc is drawn counterclockwise if sweep angle is positive, and clockwise if sweep angle is negative. Therefore, to draw an arc from twelve o'clock to six o'clock , the start angle would be 90 and the sweep angle 180.

Example:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="app6" width=300 height = 200>
</applet>
*/
public class app6 extends Applet
{
public void paint(Graphics g)
{
g.drawLine(10,10,50,50);
g.drawRect(10,60,40,30);
g.setColor(Color.red);
g.fillRect(60,10,30,80);
g.drawRoundRect(10,100,80,50,10,10);
g.fillRoundRect(20,110,60,30,5,5);
g.drawOval(60,190,200,120);
g.fillOval(110,200,100,100);
g.drawArc(30,170,80,80,10,95);
}}
```

Drawing Polygons:

It is possible to draw arbitrarily shaped figures using **drawPolygon( ) & fillPolygon( )**, show here:

void drawPolygon(int x[],int y[],int numpoints)

void fillPolygon(int x[],int y[], int numpoints)

The polygon's endpoints are specified by the coordinate pairs contained within the x and y arrays. The number of points defined by x and y is specified by numpoints. Ther are alternative forms of these methods in which the polygon is specified by a **polygon** object.

Example:

```
import java.awt.*;
import java.applet.*;
/*
<applet code = "app21" width = 300 height = 200>
</applet>
*/
```

```
public class app21 extends Applet
{
public void paint(Graphics g)
{
int xpoints[]={30,200,30,200,30};
int ypoints[]={30,30,200,200,30};
int num= 5;
g.drawPolygon(xpoints,ypoints,num);
}}
```

AWT Controls:

Controls are components that allow a user to interact with your application in various ways – for example, a commonly used control is the push button. A layout manager automatically positions components within a container.

Control Fundamentals:

The AWT supports the following types of controls:

Labels, Push buttons, Check boxes, Choice lists, Lists, Scroll bars, Text editing.

These controls are subclasses of **Component.**

Labels:

The easiest control to use is a label. A *label* is an object of type **Label,** and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. **Label** defines the following constructors:

Label( ) , Label (String str) , Label(String str, int how)

The first version creates a blank label. The second version creates a label that contains the string specified by str. This string is left-justified. The third version creates a label that contains the string specified by str using the alignment specified by how. The value of how must be one of these three constants: **Label.LEFT, Label.RIGHT, Label.CENTER.**

You can set or change the text in a label by using the **setText( )**method. You can obtain the current label by calling **getText( )**. These methods are shown here:

void setText(String str)

void getText( )

For **setText( ),** str specifies the new label. For **getText( )**, the current label is returned.

TextField

The **TextField** class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. **TextFields** is a subclass of **TextComponent. TextField** defines the following constructors:

TextField( )

TextField(int numchars)

TextField(String str)

TextField(String str, int numchars)

The first version creates a default text field. The second form creates a text field that is numchars characters eide. The third form initializes the text field with the string contained in str. The fourth form initializes text field and sets it width.

TextField (and its superclass **TextComponint)** provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call **getText( ).** To set the text, call **setText( ).** These methods are as follows:

String getText( )

void setText(String str)

Here, str is the new String.

Example:

```
import java.awt.*;
import java.applet.*;
//EXAMPLE OF LABLE AND TEXT
/*
<applet code="app9" width = 300 height = 200>
</applet>
*/
public class app9 extends Applet
{
TextField text1,text2;
Font f= new Font("TimesRoman",Font.ITALIC,20);

public void init()
{
```

```
Label one=new Label("a value");
text1=new TextField(8);
Label two=new Label("b value");
text2=new TextField(8);
add(one);
add(text1);
add(two);
add(text2);
text1.setText("0");
text2.setText("0");
}

public void paint(Graphics g)
{
g.setFont(f);
int x,y,z=0;
String s1,s2;
g.drawString("Input a number in each box",50,100);
s1=text1.getText();
x=Integer.parseInt(s1);
s2=text2.getText();
y=Integer.parseInt(s2);
z=x+y;
g.drawString("sum value is = "+z,50,175);
}}
```

Using Buttons

The most widely used control is the push button. A push button is a component that contains a label and the generates an event when it is pressed. Push buttons are objects of type **Button. Button** defines these two constructors:

Button( )

Button(String str)

The first version creates an empty button. The second creates a button that contains str as a label.

Handling Buttons:

Each time a button is pressed, an action event is generated. Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed( )** method, which is called when an event occurs. An **ActionEvent** objects is supplied as the argument to this method.

Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

// EXAMPLE OF BUTTON
/*
<applet code="app10" width =300 height = 200>
</applet>
*/

public class app10 extends Applet implements ActionListener
{
Label result;
public void init()
{
Button b1=new Button("apple");
b1.addActionListener(this);
add(b1);
Button b2= new Button("orange");
b2.addActionListener(this);
add(b2);
Button b3= new Button("banana");
b3.addActionListener(this);
add(b3);
result=new Label("                    ");
add(result);
}
public void actionPerformed(ActionEvent ae)
{
result.setText(ae.getActionCommand());
}}
```

Another one example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

// EXAMPLE OF BUTTON
/*
<applet code="app101" width =300 height = 200>
<param name=m1 value=80>
<param name=m2 value=90>
```

```
</applet>
*/
public class app101 extends Applet implements ActionListener
{
String ma1,ma2,a;
int m1,m2;
int a1,b,c1;
TextField text1=new TextField(8);
public void init()
{
ma1=getParameter("m1");
m1=Integer.parseInt(ma1);
ma2=getParameter("m2");
m2=Integer.parseInt(ma2);
Button b1=new Button("+");
b1.addActionListener(this);
add(b1);
Button b2= new Button("-");
b2.addActionListener(this);
add(b2);
Button b3= new Button("*");
b3.addActionListener(this);
add(b3);
}
public void actionPerformed(ActionEvent ae)
{
a=ae.getActionCommand();
if(a.equals("+"))
{
b=m1+m2;
}
else if(a.equals("-"))
{
b=m1-m2;
}
else
{
b=m1*m2;
}
repaint();
}
public void paint(Graphics g)
{
g.drawString("result: "+b,200,200);
}
}
```

Applying Check Boxes:

A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the checkbox class.

Checkbox supports these constructors:

Checkbox( )

Checkbox(String str)

Checkbox(String str, Boolean on)

Checkbox(String str, Boolean on, CheckboxGroup cbgroup)

Checkbox(String str,CheckboxGroup cbgroup, Boolean on)

The first form creates a checkbox whose label is initially blank. The state of the check box in unchecked.

The second form creates a check box whose label is specified by str. The state of the check box is unchecked.

The third form allows you to set the initial state of the check box. If on is **true** , the check box in initially checked; otherwise it si cleared.

The fourth and fifth forms create a check box whose label is specified by str and whose group is specified by cbgroup. If this check box is not part of a group, then cbgroup must be **null.**

Handling Check Boxes:

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. The interface defines the **itemStateChanged( )** method.an **ItemEvent** objects is supplied as the argument to this method.

CheckboxGroup:

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called radio buttons, because they act like the station selector on a car radio-only one station can be selected at any one time. Checkbox groups are objects of type **CheckboxGroup.** Only the default constructor is defined, which creates an empty group.

Choice Controls:

The **Choice** class is used to create a pop-up lists of items from which the user may choose. Thus a **Choice** control is a form of menu. When inactive, a **choise** component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made.

Choice only defines the default constructor, which creates an empty list. To add a selection to the list, call **addItem( )** or **add( ).** They have these general forms:

void addItem(String name)

void add(String name)

Here, name is the name of the item being added. Items are added to the list in the order in which calls to **add( ) or addItem( )** occur.

Handling choice list:

Each time a choice is selected, an item event is generated. This is set to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )bmethod.** An **ItemEvent** object is supplied as the argument to this method.

Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

//example of checkbox and choice
/*
<applet code="app11" width=300 height =200>
</applet>
*/

public class app11 extends Applet
{
Label o1=new Label("                                                    ");
Label o2=new Label("                                                    ");

public void init()
{

CheckboxGroup c= new CheckboxGroup();
Checkbox c1=new Checkbox("black and white",c,true);
Checkbox c2=new Checkbox("Color",c,false);
c1.addMouseListener(new check1());
c2.addMouseListener(new check2());
add(c1);
add(c2);
Choice abc=new Choice();
abc.add("onida");
abc.add("bpl");
abc.add("sumsung");
abc.add("philps");
abc.addItemListener(new ch());
add(abc);
add(o1);
add(o2);
}
class check1 extends MouseAdapter
{
public void mouseClicked(MouseEvent e)
{
o1.setText("you have selected : black and whit tv");
}
}
class check2 extends MouseAdapter
{
```

```
public void mouseClicked(MouseEvent e)
{
o1.setText("you have selected : color tv");
}
}

class ch implements ItemListener
{
public void itemStateChanged(ItemEvent e)
{
String s= (String)e.getItem();
o2.setText("you have selected " + s + " brand");
}
}
}
```

Using Lists:

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also created allow multiple selections.

To add a selection to the list, call **add( )**. it has the following two forms:

void add(String name)

void add(String name, int index)

Here, name is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by index. Indexing begins at zero. You can specify –1 to add he item to the end of the list.

Given an index, you can obtain the name associated with the item at that index by calling **getItem( )** which has this general form

String getItem(int index)

Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*
<applet code="app12" height = 200 width = 300>
</applet>
*/
```

```
public class app12 extends Applet
{
List ob=new List();
TextField tx=new TextField(10);
Button b1=new Button("add");
String count[]={"one","two","three"};
public void start()
{
Label one=new Label("text");
add(one);
add(tx);
for(int i=0;i<count.length;++i)
ob.addItem(count[i]);
add(ob);
b1.addActionListener(new Add());
add(b1);
}
class Add implements ActionListener
{
public void actionPerformed(ActionEvent e)
{
ob.addItem(tx.getText());
}}}
```

Managing Scroll Bars:

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the slider box for the scroll bar.

Scrollbar defines the following constructors:

Scrollbar( )

Scrollbar(int style)

Scrollbar(int style,int initial value, int thumbsize, int min, int max)

The first form creates a vertical scroll bar.

The second and third forms allow you to specify the orientation of the scrollbar. If style is **Scrollbar.VERTICAL,** a vertical scroll bar is created. If style is **Scrollbar.HORIZONTAL,** the scroll bar is horizontal.

In the third form of the constructor, the initial value of the scroll bar is passed in initial value. The number of units represented by the height of the thumb is passed int thumb size. The minimum and maximum values for the scroll bar are specified by min and max.

Handling Scroll Bars:

To process scroll bar events, you need to implement the **AdjustmentListerner** interface. Each time a user interacts with a scroll bar, an **Adjustmentevent** object is generated. Its **getAdjustmentType()** method can be used to determinet the type of the adjustment.

Using TextArea:

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multilane editor called **TextArea.**

TextArea is a subclass of **TextComponent**. Therefore , it supports the **getText( ), setText( )** methods described in the preceding section. **TextArea** adds the following methods:

void append(String str)

void insert(String str, int index)

The **Append( )** method appends the string specified by str to the end of the current text. Insert ( ) inserts the string passed in str at the specified index.

Example:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code = "app13" width = 300 height = 200>
</applet>
*/
public class app13 extends Applet implements AdjustmentListener
{
TextArea ta;
public void init()
{
Scrollbar sb=new Scrollbar(Scrollbar.VERTICAL,0,0,0,100);
sb.addAdjustmentListener(this);
add(sb);
```

```
ta=new TextArea(10,20);
add(ta);
}
public void adjustmentValueChanged(AdjustmentEvent ae)
{
Scrollbar sb=(Scrollbar)ae.getAdjustable();
ta.append("AdjustmentEvent :"+sb.getValue()+"\n");
}}
```

Layout Managers:

Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used.

The **setLayout( )** method has the following general form
void setLayout(LayoutManager layout obj)

FlowLayout:

FlowLayout is the default layout manager. This it the layout manager that the preceding examples have used. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right.

Constructor of flow layout is
FlowLayout(int how)
This form lets you specify how each line is aligned. Valid values for how are as follws
FlowLayout.LEFT, FlowLayout.CENTER , FlowLayout.RIGHT.

Example:

```
import java.applet.*;
import java.awt.*;
/*
<applet code = "app14" width = 300 height = 200>
</applet>
*/
public class app14 extends Applet
{
String str[]={"one","two","three","four","five","six","seven"};
public void init()
```

```
{
setLayout(new FlowLayout(FlowLayout.LEFT));
for(int i=0;i<str.length;++i)
add(new Button(str[i]));
}}
```

BorderLayout:

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are feferred to as north, south, east, and west. The middle area is called the center. **BorderLayout** defines the following constants that specify the regions:

BorderLayout.CENTER	BorderLayout.SOUTH
BorderLayout.EAST	BorderLayout.WEST
BorderLayout.NORTH	

When adding components, you will use these constants with the following form of **add( ) ,** which is defined by container.

Example

```
import java.awt.*;
import java.applet.*;

/*
<applet code="app16" width = 300 height = 20>
</applet>
*/
public class app16 extends Applet
{
public void init()
{
setLayout(new BorderLayout(5,5));
Button b1=new Button("north");
Button b2=new Button("south");
Button b3=new Button("east");
Button b4=new Button ("west");
Button b5 = new Button("center");
add(b1,"North");
add(b2,"South");
add(b3,"East");
add(b4,"West");
add(b5,"Center");
}}
```

GridLayout:

GridLayout lays out components in a two-dimensional grid. When you instantiate a GidLayout, you define the number of rows and columns. The constructors supported by gridlayout ar shown here

GridLayout(int numRows, int numColumns)

Here, this form creates a grid layout with the specified number of rows and columns.

Example:

```
import java.applet.*;
import java.awt.*;
//example of gridlayout
/*
<applet code = "app15" width = 300 height = 200>
</applet>
*/
public class app15 extends Applet
{
public void init()
{
setLayout(new GridLayout(3,4,10,10));
for(int i=1;i<=12;++i)
add(new Button(" "+i));
}}
```

Handling Events by Extending AWT Components:

To extend an AWT component, you must call the **enableEvents( )** method of **Component.** Its general form is shown here:

Protected final void enableEvents(ling eventMask)

The eventMask argument is a bit mask that defines the events to be delivered to this component. The AWTEvent class defines **int** constants for making the mask. Several are shown her:

ACTION_EVENT_MASK	ITEM_EVENT_MASK
ADJUSTMENT_EVENT_MASK	KEY_EVENT_MASK
COMPONENT_EVENT_MASK	MOUSE_EVENT_MASK
CONTAINER_EVENT_MASK	TEXT_EVENT_MASK
INPUT_METHOD_EVENT_MASK	WINDOW_EVENT_MASK

You must also override the appropriate method from one of your superclasses in order to process the event. Given lists the methods most commonly used and the classes that provide them.

Class	**processing methods:**
Button	processActionEvent( )
Checkbox	processItemEvent( )
CheckboxMenuItem	processItemEvent( )
Choice	processItemEvent( )
Component	processComponentEvent( ), processKeyEvent( ), ProcessMouseEvent( )
List	processActionEvent( ), processItemEvent( )
Scrollbar	processAdjustmentEvent( )
TextComponent	processTextEvent( )

Extending Button:

The following program creates an applet that displays a button labeled" test Button". When the button is pressed, the string "action event:", is displayed on the status line of the applet viewer or browser, followed by a count of the number of button presses.

The program has one top-level class named app18 that extends Applet. A integer variable I is defined and initialized to zero. The records the number of button pushes. The init( ) method instantiates MyButton and adds it to the applet.

MyButton is an inner class that extends Button. Its constructor uses super to pass the label of the buton to the surperclass constructor. It calls enableEvents( ) so that action events may be received by this object. When an action event is generated, processActionEvent( ) is called.

Example:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;                                                    arun

/*
<applet code="app18" width = 300 height = 200>
</applet>
*/

public class app18 extends Applet
```

```
{
MyButton one;

 int i=0;
public void init()
{
one=new MyButton("test button");
add(one);
}
class MyButton extends Button
{
MyButton(String label)
{
super(label);
enableEvents(AWTEvent.ACTION_EVENT_MASK);
}
public void processActionEvent(ActionEvent ae)
{
showStatus("action Event " + i ++);

}}}
```

Extending Checkbox:

The following program creates an applet that displays three check boxes labeled “apple”, “orange”,”mango”. When a check box is selected or deselected, a string containing the name and state of the check box is displayed on the status line of the applet viewer or browser.

The program has one top-level class named app19 that extends Applet. Its init( ) method creates three instances of Mycheckbox and adds these to the applet. Mycheckbox is an inner class that extends checkbox.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;


/*
<applet code="app19" width = 300 height = 200>
</applet>
*/

public class app19 extends Applet
```

```
{
MyCheckbox cb1,cb2,cb3;
public void init()
{
cb1=new MyCheckbox("apple ");
add(cb1);
cb2=new MyCheckbox("orange");
add(cb2);
cb3=new MyCheckbox("mango");
add(cb3);
}
class MyCheckbox extends Checkbox
{
MyCheckbox(String label)
{
super(label);
enableEvents(AWTEvent.ITEM_EVENT_MASK);
}
public void processItemEvent(ItemEvent e)
{
showStatus("Checkbox name/state: "+getLabel()+"/"+getState());
}}}
```

Extending Check Box Group:

This program reworks the preceding check box example so that the check boxes form a check box group. Thus, only one of the check boxes may be selected at any time.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*
<applet code="app20" width = 300 height = 200>
</applet>
*/

public class app20 extends Applet
{
CheckboxGroup cbg;
MyCheckbox cb1,cb2,cb3;
public void init()
{
cbg=new CheckboxGroup();
cb1=new MyCheckbox("item1 ",cbg,true);
```

```
add(cb1);
cb2=new MyCheckbox("item2",cbg,false);
add(cb2);
cb3=new MyCheckbox("item3",cbg,false);
add(cb3);
}
class MyCheckbox extends Checkbox
{
public MyCheckbox(String label,CheckboxGroup cbg, boolean flag)
{
super(label,cbg,flag);
enableEvents(AWTEvent.ITEM_EVENT_MASK);
}
protected void processItemEvent(ItemEvent e)
{
showStatus("Checkbox name/state: "+getLabel()+"/"+getState());
super.processItemEvent(e);
}}}
```

Extending Choice :

The following program creates an applet that displays a choice list with items labeled “red”, ”green”, and “blue”. When entry is selected, a string that contains the name of the color is displayed on the status line of the applet viewer or browser.

The top class name is app22 that extends applet. Its init( ) method creates a choice element and adds it to the applet. Mychoice is an inner class that extends choice. It calls enableEvents( ) so that item events may be received by this object. When an item event is generated, processItemEvent( ) is called. That method displays a string on the status line and calls processItemEvent( ) for the superclass.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
//extending choice
/*
<applet code="app22" width = 300 height = 200>
</applet>
*/
public class app22 extends Applet
```

```
{
MyChoice one;
public void init()
{
one = new MyChoice();
one.add("red");
one.add("green");
one.add("yellow");
add(one);
}

class MyChoice extends Choice
{
MyChoice()
{
enableEvents(AWTEvent.ITEM_EVENT_MASK);
}

public void processItemEvent(ItemEvent ie)
{
showStatus("Choice selection :"+getSelectedItem());
}
}}
```

Extending List:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
//extending list
/*
<applet code="app23" width = 300 height = 200>
</applet>
*/

public class app23 extends Applet
{
MyList one;
public void init()
{
one = new MyList();
one.add("red");
one.add("green");
one.add("yellow");
add(one);
}
```

```
class MyList extends List
{
MyList()
{
enableEvents(AWTEvent.ITEM_EVENT_MASK );
}
public void processItemEvent(ItemEvent ie)
{
showStatus("Item Event :"+getSelectedItem());
}}}
```

Extending Scrollbar:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
//extending scrollbar
/*
<applet code="app24" width = 300 height = 200>
</applet>
*/



public class app24 extends Applet
{
MyScrollbar one;
public void init()
{
one = new MyScrollbar(Scrollbar.VERTICAL,0,0,0,100);
add(one);
}
class MyScrollbar extends Scrollbar
{
MyScrollbar(int style,int initial, int thumb, int min,int max)
{
enableEvents(AWTEvent.ADJUSTMENT_EVENT_MASK);
}
public void processAdjustmentEvent(AdjustmentEvent ae)
{
showStatus("Adjustment Event :"+ae.getValue());
}}}
```

APPLET

Introduction:

Applets are small java programs that are primarily used in Internet computing. They can be transported over the Internet from one computer to another and run using the **Applet Viewer** or any web browser that supports java. An applet, like any application program, can do many things for us. It can perform arithmetic operations, display graphics, play sounds, accept user input, create animation and play interactive games.

We can embed applets into web pages in two ways.

One, we can write our own applets and embed them into web pages.

Second, we can download an applet from a remote computer system and then embed it into a web page.

An applet developed locally and stored in a local system in known as *local applet.*

An *remote applet* is that which is developed by someone else and stored on a remote computer connected to the internet. If our system is connected to the internet, we can download the remote applet onto our system via the Internet and run it.

In order to locate and load a remote applet, we must know the applet's address on the web. This address is known as *Uniform Resource Locator(URL)* and must be specified in the applet's HTML document as the value of CODEBASEattribute.

All Applets are subclasses of **Applet**. Thus all applets must import **java.applet.** Applets must also import **java.awt.** Recall the AWT Abstract Window Toolkit. Since all applets run in a window, it is necessary to include support for the window. Applets are not executed by the console-based java run-time interpreter. Rather, they are executed by either a web browser or an applet viewer.

Output to your applet's window is not performed by **System.out.println( ).** Rather, it is handled with various AWT methods, such as **drawstring( ),** which outputs a string to a specified x,y location. Once an applet has been compiled, it is included in an HTML file using the APPLET TAG.

The applet will be executed by a java-enabled web browser when it encounters the APPLET TAG within the HTML file. To view and test an applet more conveniently, simply include a comment at the head of your java source code file that contains the APPLET tag. This

way, your code is documented with the necessary HTML statements needed by your applet, and you can test the compiled applet by starting the applet viewer with your java source code file specified as the target.

HOW APPLETS DIFFER FROM APPLICATIONS:

Although both the applets and stand-alone applications are java programs, there are significant differences between. Applets are not full-featured application program. They are usually written to accomplish a small task or a component of a task. Since they are usually designed for use on the internet, they impose certain limitations and restrictions in their design.

Applets do not use the **main( )** method for initiating the execution of the code. Applets, when loaded, automatically call certain methods of Applet class to start and execute the applet code.

Unlike stand-alone applications, applets cannot be run independently. They are run from inside a web page using a special feature known as HTML tag.

Applets cannot read from or write to the files in the local computer.

Applets cannot communicate with other servers on the network.

Applets cannot run any program form the local computer.

Applets are restricted from using libraries from other languages such as c or c++.

The Applet Class:

The **Applet** provides all necessary support for applet execution, such as starting and stopping. It also provides methods that load and display images, and methods that load and play audio clips. **Applet** extends the AWT class **Panel.** In turn, **Panel** extends **Container,** which extends **Component.** These classes provide support for java's window-based, graphical interface.

LIFE CYCLE OF APPLET:

Init( ) --------→ Initialize a variable. Called when an applet begins execution. It is the first method called for any applet.

Start( ) -----→ After the init( ) method the program should be started. Called by By browser when an applet should start(or resume) execution. It Is automatically called after **init( )** when an applet first begins.

Paint( ) ----→drawing, writing and color creation. The **paint( )** method is called Each time your applet's output must be redrawn. **Paint( )** is also Called when the applet begins execution. This method has one Parameter of type **Graphics**. It contain the Graphics context, which describes the graphics environment. This context is used whenever output to the applet.

Stop( ) ----→Halt of the running applet. Called by browser to suspend Execution of the applet. Once stopped , an applet is restarted When the browser calls **start( ).**

Destroy( ) -> terminated. Called by browser just before an applet is terminated.

Repaint( ) →call update method.

APPLET ARCHITECTURE:

An applet is a window-based program. As such, its architecture is different from the so-called normal, console-based programs. If you are familiar with windows programming, you will be right at home writing applets.

Applets are event driven. It is important to understand in a general way how the event-driven architecture impacts the design of an applet. An applet resembles a set of interrupt service routines.

Here is how the process works. An applet waits until an event occurs. The AWT notifies the applet about an event by calling an event handler that has been provided by the applet.

The user initiates interaction with an applet – not the other way around. As you know, in a non windowed program, when the program needs input, it will prompt the user and then call some input method, such as **readLine( ). T**his is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants.

These interactions are sent to the applet as events to which the applet a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a key press event is generated. Applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

While the architecture of an applet is not as easy to understand as that of a console-based program, java's AWT makes it as simple as possible.

THE HTML APPLET TAG:

The APPLET tag is used to start an applet from both an HTML(Hyper Text Markup Language) document and from an applet viewer. (The newer OBJECT tag also works, but it will use APPLET).

An applet viewer will execute each APPLET tag that if finds in a separate window, while web browsers like Netscape Navigator, Internet Explorer, and HotJava will allow many applets on a single page. So far, we have been using only a simplified form of the APPLET tag.

The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

<APPLET
[CODEBASE = *codebase URL]*
CODE = *applet file*
WIDTH = *pixels* HEIGHT = *pixels*
[ALIGN = *alignment]* >
[<PARAM NAME = Attribute Name VALUE = Attribute value>]
[<PARAM NAME = Attribute Name2 VALUE = attribute value>]

</APPLET>

CODEBASE:

It is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file. The HTML document's URL directory is used as the CODEBASE if this attribute is not specified.

CODE:

It is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

WIDTH AND HEIGHT:

WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

ALIGN:

It is an optional attribute that specifies the alignment of the applet. This attributes is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

PARAM NAME AND VALUE:

The PARAM tag allows you to specify applet specific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method.

HANDLING BROWSERS:

The best way to design your HTML page to deal with such browsers is to include HTML text and markup within your <applet></applet> tags. If the applet tags are not recognized by your browser, you will see the alternate markup. If java is available, it will consume all of the markup between the <applet></applet> tags and disregard the alternate markup.

Example of Applet program:

```
import java.applet.*;
import java.awt.*;
/*
<applet code="app1.java" height = 400 width = 200 >
</applet>
*/
public class app1 extends Applet
{
public void paint(Graphics g)
{
g.drawString("hello applet program",30,30);
}}
```

In this program save app1.java. and compile javac app1.java. Then run this program by this command : appletviewer app1.java. The it will create a applet window and print the message “hello applet program” .

Example two:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="app2" width =400 height = 50>
</applet>
*/
public class app2 extends Applet
{
String msg;
public void init()
{
setBackground(Color.cyan);
setForeground(Color.red);
}
public void start()
{
msg="arun";
}
public void paint(Graphics g)
{
msg+="kumar ";
g.drawString(msg,130,150);
}}
```

PASSING PARAMETERS IN APPLET:

The APPLET tag in HTML allows you to pass parameters to your applet. To retrieve a parameter, use the **getParameter( )** method. It returns the value of the specified parameter in the form of **String** object. Thus, for numeric and **Boolean** values, you will need to convert their string representations into their internal formats. Here the example of passing parameters

```
import java.awt.*;
import java.applet.*;
/*
<applet code="app3" width =400 height = 500>
<param name=name value="name :saran">
<param name=no value = 100>
<param name=m1 value=80>
<param name=m2 value=90>
</applet>
*/
public class app3 extends Applet
{
Font f=new Font("TimesRoman",Font.ITALIC,20);
String name;
int no,m1,m2,tot;
public void init()
{
name=getParameter("name");
String na,ma1,ma2;
na=getParameter("no");
no=Integer.parseInt(na);
ma1=getParameter("m1");
m1=Integer.parseInt(ma1);
ma2=getParameter("m2");
m2=Integer.parseInt(ma2);
tot=m1+m2;
}
public void paint(Graphics g)
{
g.setFont(f);
g.setColor(Color.red);
g.drawString(name,50,50);
g.drawString("number :"+no,50,80);
g.drawString("mark1  :"+m1,50,110);
g.drawString("mark2  :"+m2,50,140);
g.drawString("total  :"+tot,50,170);
}}
```