INTRODUCTION TO JAVA

Java was conceived by James Gosling, at Sun Microsystems Inc., in 1991. It took 18 months to develop the first working version. This language was initially called "Oak", but was renamed "Java" in 1995. The primary motivation of Java was the need for a platform independent language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.

Java is a general purpose, current, class-based, object oriented program language. It incorporates many features of C and C ++ but is designed and organised rather differently with a number of feature of C and C ++ debted, and a new feature from other program language.

Java is strongly typed language. This specification clearly distinguishes between the compile time errors that can and must be detected at compile time, and those that occur at run time.

Java Milestones

23rd January 1996 -> Sun releases JDK 1.0
19th Feb 1997 -> Sun releases JDK 1.1 (AWT, inner class, java beans, JDBC, RMI, Reflection API, JIT (just in time) compiler.
8th Dec 1998 -> Sun releases JDK 1.2 (strictfp keyword, swing, collection framework) (Codename **Playground**)
1999 -> Sun releases J2EE (Enterprise Edition)
8th May 2000 -> Sun releases JDK 1.3(Codename **Kestrel**)
6th Feb 2002 -> Sun releases JDK 1.4(Codename **Merlin**)
30th Sep 2004 -> Sun releases JDK 1.5(Codename **Tiger**)
11th Dec 2006 -> Sun releases JDK 1.6 (Codename **Mustang)**
28th July 2011 -> Sun releases JDK 1.7 (Codename **Dolphin)**
18th Mar 2014 -> Sun releases JDK 1.8

JAVA BUZZWORDS/ FEATURES

Java provides many features which are also known as Java buzzwords. These are:

1)Simple

Java was designated to be very simple and easy to learn. The syntax of Java has been kept nearer to C++ so that the usage of Java does not require extensive training programs to be undertaken. However, in Java the infrequently used, complex features of C++ have not been included like operator overloading, multiple inheritance etc. It has no pointer concept. Eliminates much redundancy (e.g. no structs, unions). It contains no goto statement. Added features to simplify: Garbage collection, so the programmer won't have to worry about storage management, which leads to fewer bugs. A rich predefined class library. Thus, a programmer aware of the various object-oriented concepts can easily develop applications in Java.

2) Object Oriented

Simply stated, object oriented design is a technique that focuses design on the data and on the interfaces rather than modularization of the functionalities or the tools used to develop them. Java uses object-oriented concepts as for basis for S/W design. Java provides a clean and efficient object-based development platform.

3) Robust

It provides extensive compile-time checking followed by a second level of runtime checking. The most common problems in programming languages are related to memory management and exception handling. The memory management model of Java does not allow the creation of pointers. Java has a true array which allows subscript checking to be performed. Thus, Java programmers need not worry about freeing or corrupting memory as the programs cannot overwrite the end of a memory buffer. Also Java has automatic garbage collection once the memory is no longer required.

4) Secure

Java is designed to be used in networked and distributed environments where security is of paramount importance. Java supports the creation of applications that cannot be invaded from outside. On the other side, Java programs are executed in their own environment and do not go outside these boundaries unless They are authorized to do so. The authentication techniques are based on the public key encryption method.

5) Architecture Neutral

Java was designed to support applications on heterogeneous network environments composed of a variety of processors, the operating system architectures, and multiple programming language interfaces. To enable a Java application to execute anywhere on the network, the Java compiler generates the bytecode instructions which are not dependent upon a particular computer architecture. These instruction are then interpreted on any machine and translated into the native machine code on the fly by the Java runtime.

6) Portable

Besides being architecturally Uneutral, Java is strict in its definition of the basic language. Unlike C or C++, there are no "implementation dependent" aspect of the specification. The size of the primitive data types are specified as is the behaviour of the arithmetic on them. For example, "int" always means a 32 bit integer. Thus, java programs are the same on any platform. There are no data type incompatibilities across the hardware and software architectures.

7) **Interpreted**

Java bytecode is not directly executed by the system, because Java is interpreted. The Java compiler generates *byte-codes*, rather than native machine code. To actually run a Java program, you use the Java interpreter to execute the compiled byte-codes. Java byte-codes provide an architecture-neutral object file format. The code is designed to transport programs efficiently to multiple platforms.

- rapid turn-around development
- Software author is protected, since binary byte streams are downloaded and not the source code

8) **Multithreaded**

Java was designed to meet the real world of creative, interactive, networked programs. Java's multithreading capability provides the means to build applications with many concurrent threads of activity. The multithreading feature of Java has various sophisticated synchronization primitives. Moreover, Java's high level system libraries have been written to be thread safe, i.e., the functionality provided by the libraries is available without conflict to multiple concurrent threads of execution.

9) **Dynamic**

While the Java compiler is strict in its static checking during compile time, the language and run time systems are dynamic during linking and loading stages. Classes are linked only as needed. New code modules can be linked in on demand from a variety of sources, even from sources across the network in a large number. By making these interconnections between modules later, libraries can freely add new methods and instance variables without any effect on their client.

10) Distributed

Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs. Java applications can open and and access remote objects on internet as easily as they can do in a local system. This enables multiple programmers at multiple remote applications to collaborate and work together on a single project.

11) Platform Independent: When Java Code is compiled a byte code is generated which is independent of the system. This byte code is fed to the JVM (Java Virtual Machine) which resides in the system. Since every system has its own JVM, it doesn't matter where we compile the source code. The byte code generated by the compiler can be interpreted by any JVM of any machine. Hence it is called Platform independent Language.

Java's byte code are designed to be read and interpreted in exactly same manner on any computer hardware or operating system that supports Java Runtime Environment.

When we write the program for any programming language it is called (source program), and it will have the extension depending upon the language.

In Java, when we write the program, we will have the ".java" file. And when it is compiled, we will get the ".class" file. The ".class" file is executed by the Java Virtual Machine (JVM). If we have the JVM, we can execute the java program anywhere under operating system. That means, that Java is PLATFORM INDEPENDENT.

JAVA LIBRARIES

The complete Java system includes a number of libraries of utility classes and methods of use to developers in creating multi-platform applications. It is also called as **Application Programming Interface (API)** These libraries are:

1) import java.lang.* -> A collection of classes and methods required for implementing basic features of java.

2) import java.io.* -> A collection of classes required for I/O manipulation.

3) import java.util.* -> A collection of classes to provide utility functions such as date and time functions.

4) import java.net.* -> A collection of classes for communicating with other computers via Internet.

5) import java.awt.* -> A Abstract Window Tool Kit package contains classes that implements platform-independent graphical user interface.

6) import java.applet.* -> This include a set of classes that allows us to create Java applets.

Object-Oriented Concept

Object-oriented programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and

functions that can be used as templates for creating copies of such modules on demand.

OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it and protect it from unintentional modification by other functions. OOP allows us to decompose the problem into a number of entities called object and then build data and functions (known as methods in Java) around these entities. The combination of data and method make up an object.

Some of the features of Object-oriented programming are :

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data and structure are designed such that they characterized the object.
- Methods that operate on the data of an object are tied together in a data structure.
- Data is hidden and cannot be accessed by external functions.
- Object may communicate with each other through methods.
- New data and methods can easily added whenever necessary.
- Follows bottom-up approach in program design.

1) **Class ->** A class is a way of binding the data and associated methods in a single unit. Whatever methods we write in the class are known as member methods.

Note :- whatever methods does not come under scope of the class are known as non-member methods. Java does not supports non-member methods.

Note ->Whenever we design a class there is no memory space for the data members of the class.

Classes are building blocks of java application whenever we want to develop any application in java that should be developed w.r.to class only.

2) **object ->**

a) A class variable is known as an object.

b) Grouped item is known as an object.

c) Instance of a class is known as an object.

Instance is the mechanism of allocating sufficient amount of memory space for the data members of the class.

d) Blueprint of the class is known as the object.

e) It is a logical run-time entity.

3) **Data abstraction ->** It is mechanism of retrieving he essential details by hiding the background details.

Data abstractions are of three types :-

a) Physical level abstraction :- When abstraction is said to be P.L.A if and only if it deals with physical architecture of the data. (design level).

b) Conceptual/Logical level abstraction :- It is one which deals with actual content of the application without dealing with its physical organization.

c) View level abstraction :- It is one in which the data accessed by many viewers. (Browser client/ Web client).

4) **Data encapsulation :-** It is the mechanism of wrapping up of data and associated methods together in a single unit. The Purpose of data encapsulation is to achive the security when we transmit the data from client to the server and server to the client. It is also used for achieving information hiding or data hiding.

5) **Inheritence** -> Inheritence is the mechanism of obtaining the properties from one class to another class.

The class which is giving the property is known as base class or super class or parent class.

The class which is taking the property is known as sub class or child class.

Note : The process of inheritance is also known as sub classing or extendable classes or reusability.

Advantages

- Redundancy of the code is reduced.
- Investment cost towards the project is reduced.
- It leads to less storage cost.

Sub class :- A class is said to be a sub class if and only if contains some features on its own plus it takes some property from base class.

6) Polymorphism :- It is a mechanism of representing one thing in many forms.

There are two types of polymorphism. They are compile time polymorphism and run time polymorphism. Compile time polymorphism is not supported by java. Java supports only run-time polymorphism.

Run-time polymorphism is also known as Dynamic method dispatch (DMD).

7) Dynamic binding :- It is a mechanism of binding an appropriate version of the method with super class object at run-time.

The purpose of dynamic binding is to solve the total problem of static binding.

Static binding -> It is one the memory space is created at compile time. When we use static binding we get the following problems they are :-

a) Waste of memory space.
b) Loss of data.
c) Overlapping.

Overview of Java Language

Java is a general-purpose, object-oriented programming language. We can develop two types of java programs :

a) Stand-alone applications
b) Web applets

Stand-alone applications are programs written in java to carry out certain tasks on a stand-alone computer. In fact, Java can be used to develop programs for all kinds of applications. Executing a stand-alone Java program involves two steps

1. Compiling source code into bytecode using javac compiler.
2. Executing the bytecode program using java interpreter.

Source code --------- byte code --------- machine code
javac java

Simple Java program

```
class Simple
  {
   public static void main(String args[])
    {
    System.out.println("Hello Java");
    }
 }
```

1) class is a keyword and declares that a new class definition follows. Simple is a Java identifier that specifies the name of the class to be defined.

2) Every class definition in Java begins with an opening brace "{" and ends with a matching closing brace "}", appearing in the last line.

3) Every java application program must include the main() method. This is the starting point for the interpreter to begin the execution of the program. A Java application can have any number of classes but only one of them must include a main method to initiate the execution.

Note :- Java applets will not use the main method at all.
public -> The keyword public is an access specifier that declares the main method as unprotected and therefore making it accessible to all other classes.

static -> It declares this method as one that belongs to the entire class and not a part of any objects of the class. The main must always be declared as static since the interpreter uses this method before any objects are created.

void -> The Type modifier void states that the main method does not return any value.

String args[] -> It declares a parameter named args, which contains an array of objects of the class type String.
System.out.println() -> The println method is a member of the out object, which is a static data member of System class. The method println always appends a newline character to the end of the string. This means that any subsequent output will start on a new line. Every Java statements must end with a semicolon.

Comments

Java permits both the single-line comments and multi-line comments. The single-line comments begin with // and end at the end of the line. For multi-line comments starting with a /* and ending with a */ .

Java Program Structure

Documentation Section -> Suggested
Package Statement -> Optional
Import Statements -> Optional
Interface Statements -> Optional
Class Definitions -> Optional
main method class
{ -> Essential
main method definition
}

Documentation Section

The documentation section comprises a set of comments lines the name of the program, the author and other details, which the programmer would like to refer to at a later stage.

Package Statement

The first statement allowed in a java file is a package statement. This statement declares a package name and informs the compiler that the classes defined here belong to this package. For example,

package student;

Import Statements

It is used to import a package and all its classes.

Interface Statements

An interface is like a class but includes a group of method declarations. This is also an optional section and is used only when we wish to implement the multiple inheritance feature in the program.

Class Definitions

A Java program may contain multiple class definitions. Classes are the primary and essential elements of a Java program. These classes are used to map the objects of real-world problems.

Main Method Class

Since every Java stand-alone program requires a main method as its starting point, this class is the essential part of a java program. A simple Java program may contain only this part. The main method creates objects of various classes and establishes communications between them. On reaching the end of main, the program terminates and the control passes back to the os.

WAP to add two numbers.

```
class add
  {
  public static void main(String args[])
   {
   int a=5;
   int b=10,c;
   c=a+b;
   System.out.println("Add="+c);
```

```
  }
}
```

Use of Math functions

```
import java.lang.Math;
```

The purpose of this statement is to instruct the interpreter to load the Math class from the package lang.

WAP to calculate the square root of number.

```
import java.lang.Math;
class sqroot
  {
  public static void main(String args[])
    {
     double x=5;
     double y;
     y = Math.sqrt(x);
     System.out.println("y=" + y);
      }
  }
```

WAP to Calculate the Power of any number.

```
Import java.lang.Math;
class power
  {
  public static void main(String args[])
  {
  int num=5;
  int b;
  b=Math.pow(num,4);
  System.out.println("Power of number=" +b);
 }
}
```

Write a Java program to swap two numbers .

a) with taking third variable

```
class swap
 {
  public static void main(String args[])
   {
   int a,b,c;
   a=10;
   b=12;
   c=a;
   a=b;
   b=c;
   System.out.println("After swapping");
```

```
    System.out.println("a=" +a);
    System.out.println("b=" +b);
  }
  }
```

b) without taking third variable

```
class swap
 {
  public static void main(String args[])
   {
   int a,b,c;
   a=10;
   b=12;
   a=a+b;
   b=a-b;
   a=a-b;
   System.out.println("After swapping");
   System.out.println("a=" +a);
   System.out.println("b=" +b);
  }
}
```

Java Tokens

Smallest individual units in a program are known as tokens. The compiler recognizes them for building up expressions and statements.

In Simple terms, a Java program is a collection of tokens, comments and white spaces. Java language includes five types of tokens. They are:

- Reserved Keywords
- Identifiers
- Literals
- Operators
- Seperators

Java Character Set

Java characters are defined by the Unicode Character Set. The Unicode is a 16-bit character coding system.

Keywords

Keywords are identifiers, such as public, static and class that have a special meaning inside Java source code, and outside of comments and Strings. Keywords are reserved for their intended use and cannot be used by the programmer for variable, or method names. Some of the reserved keywords in

boolean	continue	default	do	double	else
extends	final	finally	break	byte	case
for	goto	if	implements	import	interface
native	new	null	package	protected	public
return	instance of	int	catch	char	short
static	super	switch	synchronized	this	throw
throws	class	transient	try	void	volatile
while	abstract				

Identifiers in Java

Identifiers are the names of **variables, methods, classes, packages** and **interfaces.** Identifiers must be composed of **letters, numbers,** the **underscore_** and the **dollar sign $.** They cannot contain white spaces. Identifiers may only begin with a letter, the underscore or a dollar sign. we cannot begin a variable name with a number. All variable names are case sensitive.

for example:

My **Variable** is not the same as my **Variable.**

There is no limit to the length of a Java variable name.

Identifiers must be meaningful, short enough to be quickly and easily typed and long enough to be descriptive and easily read. Java developers have followed some naming conventions.

1. Names of all public methods and instance variables start with a leading lowercase letter.

Ex- average, sum

2. When more than one words are used in a name, the second and subsequent words are marked with a leading uppercase letters.

Ex : dayTemperature
totalMarks

3. All private and local variables use only lowercase letters combined with underscores.

Ex : length
batch_strength

4. All classes and interfaces start with a leading uppercase letter (and each subsequent word with a leading uppercase letter).

Ex : Student
HelloJava
Vehicle

5. Variables that represent constant values use all uppercase letters and underscores between words.

Ex : TOTAL
F_MAX

Literals/Constants

Literals are pieces of Java source code that indicate explicit values. For instance, Hello World! Is a String literal. Java has five kinds of literals: String, Character, Number, Float and Boolean.

String Literals

The string literal is always enclosed in double quotes. Java uses a **String** class to implement strings, whereas C and C++ use an array of characters. For example:
"Hello World!"
String message = "Hello World";

Character Literals

Character literals are similar to String literals except they are enclosed in single quotes and must have exactly one character. For example 'c' is a character literal. A backslash is used to denote the non-printing characters such as

Description	**Escape Sequence**
Line feed	**\n**
Carriage Return	**\r**
Horizontal tab	**\t**
Backslash	\\
Single quote	**\'**
Double quote	**\"**

Boolean Literal

A Boolean literal can have either of the values: **true or false**. They do not correspond to the numeric values, 1 and 0, as in C and C++.

Numeric Literals

An integer constant refers to a sequence of digits. There are three types of integers namely, decimal integer, octal integer and hexadecimal integer.

Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices and so on. These quantities are represented by numbers containing fractional parts like 17.548, such numbers are called real (or floating) constants. e.g, 0.0083, -0.75, 423.87

A real number may also be expressed in exponential (or scientific) notation. For e.g, 215.65 may be written as 2.1565e2. e2 means multiply by 10^2. mantissa e exponent. e.g, 7500000000= 7.5E9 or 75E8

-0.000000368 = -3.68E-7

OPERATORS

Operators are used to build expressions. They are described here in several related categories. Operators can be broadly categorised as:

a) **Relational and equality operators**

Relational operators are binary operators that require two operands to work on. The operands can be constants, variables, or expression. The operators are as follows:

> greater than

>= greater than, or equal to
< less than
<= less than, or equal to
! Boolean NOT
!= not equal to

b) **Assignment Operators**

= assignment
^= bitwise XOR and assign
&= bitwise AND and assign
%= take remainder and assign
= subtract and assign
*= multiply and assign
/= divide and assign
|= bitwise OR and assign
>>= shift bits right with sign extension and assign
<<= shift bits left and assign
>>>= unsigned bit shift right and assign

c) **Arithmetic**

- subtraction
x multiplication
/ division
+ addition
% modulo

d) **Bitwise**

| bitwise OR
^ bitwise XOR
& bitwise AND
>> right shift
<< left shift
~ bitwise NOT
>>> unsigned bit shift right

e) Logical

&& (Logical AND)
|| (Logical OR)
! (Logical NOT)

f) Increment and Decrement Operators

++ , --

The operator ++ adds 1 to the operand, while – substracts 1.

Both are unary operators and takes the following form :

++m ; or m++
--m ; or m - -
++m is equivalent to m=m + 1 or m+=1
--m is equivalent to m=m-1 or m- = 1

g) Conditional operator : A ternary operator " ?: " is available in Java to construct conditional expressions of the form

exp1 ? exp2 : exp3

where exp1, exp2 and exp3 are expressions.

The operator ?: works as follows : exp1 is evaluated first. If it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression.

e.g, a = 10;
b = 15;
x=(a>b) ? a:b;

x will be assigned the value of b.

-> if (a>b)
x=a
else
x=b

h) Special Operator

Java supports some special operators of interest such as **instanceof** operator and **member selection operator(.).**

i) instanceof operator -> The instanceof is an object reference operator and returns true if the object on the left-hand side is an instance of the class given on the right-hand side. This operator allows us to determine whether the object belongs to a particular class or not.

Example : person instanceof student

is true if the object person belongs to the class student; otherwise it is false.

Dot Operator

The dot operator (.) is used to access the instance variables and methods of class objects. Examples:

Person.age // Reference to the variable age
Person1.salary() // Reference to the method salary

Seperators

Seperators are symbols used to indicate where groups of code are divided and arranged. They basically define the shape and functions of our code.

a) Parenthesis() -> Used to enclose parameters in method definition and invocation, also used for defining precedence in expression, containing expressions for flow control, and surrounding cast types.

b) braces {} -> Used to contain the values of automatically initialized arrays and to define a block of code for classes, methods and local scopes.

c) brackets[ ] -> Used to declare array types and for dereferencing array values.

d) semicolon ;-> Used to separate statements.

e) period . -> Used to separate package names from sub-packages and classes; and also used to separate a variable or method from a reference variable.

Implementing a Java Program

Implementation of a Java application program involves a series of steps. They include

1) Creating the program
2) Compiling the program
3) Running the program

1) Creating the program

We can create a program using any text editor.

```
class Test
  {
  public static void main(String args[])
    {
    System.out.println("Welcome to Java");
    }
 }
```

We must save this program in a file called Test.java ensuring that the filename contains the class name properly. This file is called the source file. Note that all java source files will have the extension java. Note also that if a program contains multiple classes, the file name must be the classname of the class containing the main method.

2) Compiling the Program

To compile the program, we must run the Java Compiler javac, with the name of the source file on the command line.

Javac Test.java

If everything is OK, the javac compiler aerates a file called Test.class containing the bytecodes of the program. Note that the compiler automatically names the bytecode file as

<classname> . class

3) Running the program

We need to use the Java interpreter to run a stand-alone program. At the command prompt, type

java Test

Now, the interpreter looks for the main method in the program and begins execution from there.

Java interpreter reads the bytecode files and translates them into machine code for the specific machine on which the Java program is running.

Java Virtual Machine

The Java virtual machine is a specification for an abstract computer. JVM consists of a class loader and a java interpreter that executes the bytecode. The class

loader loads class files from both the java program and java API for execution by the java interpreter. The Java interpreter may be a S/W interpreter that interprets the bytecode one at a time or, it may be a just-in-time compiler that turns the architectural neutral bytecode into native machine language for the host computer. The Java interpreter may be implemented in H/W.

Command Line Arguments

In command line arguments input provided at the time of execution. Command line arguments are parameters that are supplied to the application program at the time of invoking it for execution.

public static void main(string args[])

Here args is declared as an array of strings (known as string objects). Any arguments provided in the command line (at the time of execution) are passed to the array args as its elements. We can simply access the array elements and use them in the program as we wish. For example,

Java Test BASIC FORTRAN C++ Java

The command line contains for arguments. These are assigned to the array args

BASIC -> args[0]
FORTRAN -> args[1]
C++ -> args[2]
Java -> args[3]

The individual elements of an array are accessed by using an index or subscript like args[i]. The value of i denotes the position of the elements inside the array.

```
class ComTest
  {
  public static void main(String args[])
    {
    int count, i=0;
    String str;
    count=args.length;
    System.out.println("Number of arguments =" +count);
    while(i<count)
       {
       str=args[i];
        i=i+1;
        System.out.println(i + " : " + " Java is" + string + " ! ");
        }
     }
  }
```

Java ComTest Simple Distributed Robust Secure Portable

1) WAP to enter two number and add using command line argument

```
class add
   {
  public static void main(String args[])
    {
    int a,b,sum;
    a=Integer.parseInt(args[0]);
    b=Integer.parseInt(args[1]);
    sum=a+b;
    System.out.println("sum=" +sum);
    }
}
```

java add 10 20

Addition two float values

```
a=Float.valueOf(args[0]).floatValue();
b=Float.valueOf(args[1]).floatValue();
```

Addition of two double values

```
a=Double.parseDouble(args[0]);
b=Double.parseDouble(args[1]);
```

WAP to addition of n numbers using Command-line arguments

```
class sumn
  {
  public static void main(String args[])
    {
    int i,sum=0;
    for(i=0;i<args.length;i++)
      {
      sum=sum+Integer.parseInt(args[i]);
      }
  System.out.println("sum=" +sum);
  }
}
```

java sumn 10 20 30 40 50

Variables

A Variable is an identifier that denotes a storage location used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during the execution of the program. A Variable name can be chosen by the programmer in a meaningful way.

Variable names may consist of alphabets, digits, the underscore(_) and dollar characters.

1. They must not begin with a digit.

2. UpperCase and lowercase are distinct. This means that the variable Total is not the same as total or TOTAL.
3. It should not be a keyword.
4. White space is not allowed.
5. Variable names can be of nay length.

In Java, Variables are the names of storage locations. Variable declaration does three things :

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.
3. The place of declaration (in the program) decides the scope of the variable.

Syntax :

Type variable1, variable2, ……. variableN;

Ex :- int count;
float x,y;
double pi;
byte b;
char c1,c2,c3;

Giving Values to Variables

A Variable must be given a value after it has been declared but before it is used in an expression. This can be achieved in two ways :

1. By using an assignment statement
2. By using a read statement.

1. Assignment Statement

A Simple method of giving value to a variable is through the assignment statement as follows :

variableName=value;

For example :

initial=0;
finalvalue=100;
yes='x';

We can also string assignment expression as follows :

x=y=z=0;

It is also possible to assign a value to a variable at the time of its declaration.

Example : int a=50;

The process of giving initial values to variables is known as the initialization.

Read Statement

We may also give value to variables interactively through the keyboard using the readLine() method.

```
import java.io.*;
class read
```

```
{
public static void main(String args[]) throws IOException
  {
  int a,b,sum;
  BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
  System.out.println("Enter the first number :");
  a=Integer.parseInt(br.readLine());
  System.out.println("Enter the second number :");
  b=Integer.parseInt(br.readLine());
  sum=a+b;
  System.out.println("sum=" +sum);
  }
}
```

Scope of Variables

Java variables are actually classified into three kinds :

1. instance variables,
2. class variables, and
3. local variables

1. **instance variables ->** Instance variables are created when the objects are instantiated and therefore they are associated with the objects. They take different values for each object.

2. **class variables->** class variables are global to a class and belong to the entire set of objects that class creates. Only one memory location is created for each class variables.

3. **local variables ->** Variables declared and used inside methods are called local variables. They are called so because they are not available for use outside the method definition. Local variables can also be declared inside program blocks that are defined between an opening brace { and a closing brace }. These variables are visible to the program only from the beginning of its program block to the end of the program block. When the program control leaves a block, all the variables in the block will cease to exist. The area of the program where the variable is accessible (i.e, usable) is called its scope.

Symbolic Constants

A constant is declared as follows :

Syntax :- final type symbolic-name=value;

Examples :

```
final int STRENGTH=100;
final int PASS_MARK=50;
final float PI=3.14159;
```

Note :

1. Symbolic names take the same form as variable names. But, they are written in CAPITALS to visually distinguish them from normal variable names.

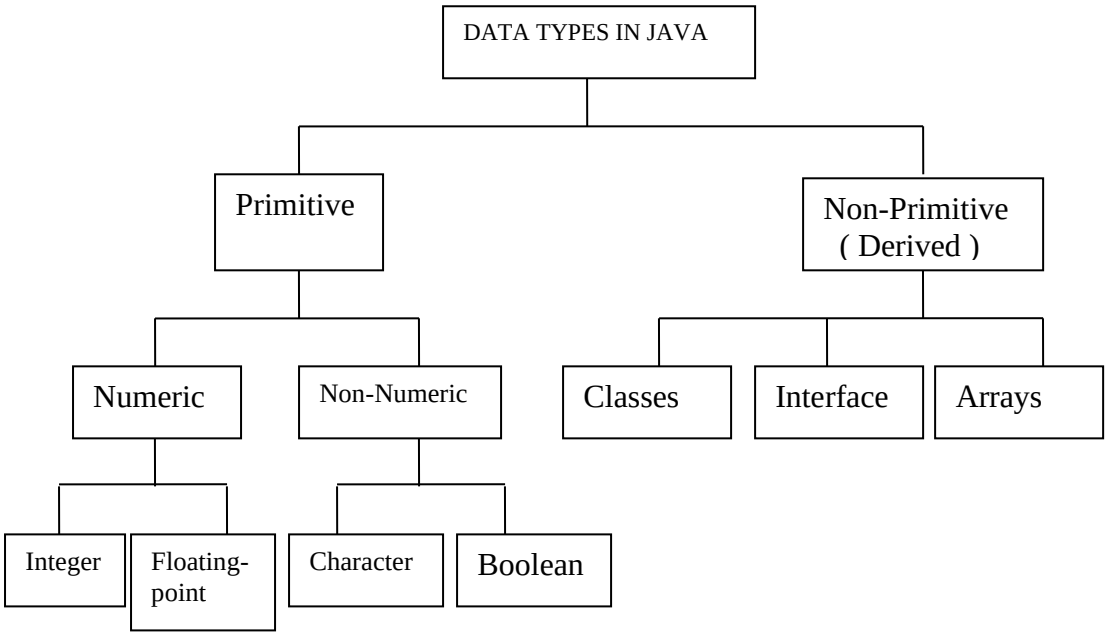
2. After declaration of symbolic constants, they should not be assigned any other value within the program by using an assignment statement. For example, STRENGTH=200; is illegal.
3. Symbolic constants are declared for types.

Data Types

Data types specify the size and type of values that can be stored. Java language is rich in its data types. The variety of data types available allow the programmer to select the type appropriate to the needs of the application.

Java supports two classes of data types :-

1. Primitive data types
2. Non-Primitive(Derived) data types



1. Integer Types

Integer types can hold whole numbers such as 123,-96. Java supports four types of integers. They are byte, short, int, and long. Java does not support the concept of unsigned types and therefore all Java values are signed meaning they can be positive or negative.

Type	Size	Minimum value	Maximum value
byte	One byte	-128	+127
short	Two byte	-32,768	+32,767
int	Four bytes	-2,147,483,648	+2,147,483,647
long	Eight bytes	……..	……

2. Floating Point Types

It holds fractional parts such as 27.59 and -1.375 (known as floating point constants). There are two kinds of floating point storage in Java.

The float type values are single-precision numbers while the double types represent double-precision numbers.

Floating point numbers are treated as double-precision quantities. To force them to be in single-precision mode, we must append f or F to the numbers.

Example : 1.23f
7.569f

Type	Size	Minimum value	Maximum value
float	4 bytes	3.4e-038	3.4e+038
double	8 bytes	1.7e-308	1.7e+308

Double-precision types are used when we need greater precision in storage of floating point numbers. All mathematical functions, such as sin, cos and sqrt return **double** type values.

Character Type

In order to store character constants in memory, Java provides character data type called char. It holds 2 bytes of memory.

Boolean Type

Boolean type is used when we want to test a particular condition during execution of the program. There are only two values that a boolean type can take : true or false. Boolean type is denoted by the keyword **boolean** and uses only 1 bit of storage.

Type Casting

To store a value of one type into a variable of another type. In such situations, we must cast the value to be stored by proceeding it with the type name in parentheses.

Syntax : type variable1=(type) variable2;

The process of converting one data type to another is called casting.

Examples :

```
int m=50;
byte n=(byte) m;
long count=(long)m;
```

Casting into a smaller type may result in a loss of data. Similarly, the **float** and **double** can be cast to any other type except boolean. Casting a floating point value to an integer will result in a loss of the fractional part.

Automatic Conversion

For some types, it is possible to assign a value of one type to a variable of a different type without a cast. Java does the conversion of the assigned value automatically. This is known as automatic type conversion. Automatic type conversion is possible only if the destination type has enough precision to store the source value. For example,

```
byte b=75;
int a=b;
```

The process of assigning a smaller type to a larger one is known as widening or promotion and that of assigning a larger type to a smaller one is known as narrowing. Narrowing may result in loss of information.

Decision Statements

The decision statement decides the statement to be executed after the success or failure of a given condition. The decision-making statement checks the given condition and then executes its sub-block.

Java language possesses decision-making capabilities supporting the following statements ;-

1. if statement
2. switch statement
3. conditional operator statement

These statements are popularly known as decision –making statements. Since these statements control the flow of executing, they are also known as control statements.

a) The if statement
b) The if-else statement
c) Nested if-else statement
d) The if-else-if ladder statement

a) The if statement

Java uses the keyword if to execute a set of command lines or one command line when the logical condition is true. It has only one option. The set of command lines or command lines are executed only when the logical condition is true.

Syntax :

```
if(condition)
    statements;
```

e.g,

WAP to check whether the entered number is less than 10? if yes, display the same.

```
class num
   {
   public static void main(String args[])
    {
    int v;
    if(v<10)
       {
        System.out.println("Number entered is less than 10");
         }
   }
}
```

2. The if..else statement

The if..else statement takes care of true as well as false conditions. It has two blocks. One block is for if and it is executed when the condition is true. The other block is of else and it is executed when the condition is false. The else statement cannot be used without if. No multiple else statements are allowed with one if.

Syntax :

```
if(condition)
   statement1;
else
   statement2;
```

WAP to enter number and check no. is even or odd.

```
import java.io.*;
class odd
  {
  public static void main() throws IOException
     {
    int num;
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
    System.out.pritln("Enter number :");
     num=Integer.parseInt(br.readLine());
     if(num%2==0)
     System.out.pritln("even number");
     else
     System.out.println("odd number");
     }
   }
```

Nested if..else

```
if(condition)
    statement;
else
  {
  if(condition)
     statement;
  else
   {
   statement;
    }
}
```

WAP to enter three numbers and find largest.

```
import java.io.*;
```

```
class large
 {
public static void main(String args[]) throws IOException
    {
    int a,b,c;
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
   System.out.println("Enter three number :");
   a=Integer.parseInt(br.readLine());
   b=Integer.parseInt(br.readLine());
   c=Integer.parseInt(br.readLine());
   if(a>b)
      if(a>c)
       {
       System.out.println("a is greater");
       }
  else
     {
       System.out.println("c is greater");
     }
else if(b>c)
   {
       System.out.println("b is greater");
    }
else
  {
  System.out.pritln("c is greater");
 }
}
}
```

The if..else if ladder

Syntax :

```
if(condition1)
     statement;
else if(condition2)
     statement;
else if(condition3)
     statement;
…….
else
  statement;
```

e.g,

WAP to enter five subjects of marks and calculate sum and average.
If avg>=60 print "First division"
Avg>=45 and avg<60 "Second division
Avg>=30 and avg<45 "Third division
Else fail

```
import java.io.*;
class first
 {
public static void main(String args[]) throws IOException
 {
int phy,che,math,bio,eng;
 BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter Five subject of marks :");
phy=Integer.parseInt(br.readLine());
che=Integer.parseInt(br.readLine());
math=Integer.parseInt(br.readLine());
bio=Integer.parseInt(br.readLine());
eng=Integer.parseInt(br.readLine());
float sum,avg;
sum=phy+che+math+bio+eng;
avg=sum/5;
if(avg>=60)
    System.out.println("First division");
else if(avg>=45 && avg<60)
    System.out.println("Second division");
else if(avg>=30 && avg<45)
    System.out.println("Third division");
else
     System.out.println("Fail");
  }
}
```

The break statement

The keyword break allows the programmers to terminate the loop. The break skips from the loop or block in which it is defined. The control then automatically goes to the first statement after the loop or block. The break can be associated with all conditional statements.

We can also use break statements in the nested loops. If we use break statement in the innermost loop then the control of the program is terminated only from the innermost loop.

The continue statement

The continue statement is exactly opposite to break. The continue statement is used for continuing next iteration of loop statements. When it occurs in the loop it does not terminate, but it skips the statements after this statement. It is useful when we want to continue the program without executing any part of the program.

Difference between break and continue

break	**continue**
1) exits from current block or loop	loop takes next iteration
2) control passes to next statement	control passes at the beginning of loop.
3) terminates the program	never terminates the program.

The switch statement

Java has a built-in multiway decision statement known as a switch. The switch statement tests the value of a given variable(or expression) against a list of case values and when a match is found, a block of statements associated with the case is executed.

Syntax :-

```
switch (expression )
   {
 case value-1:
       block-1;
        break;
  case value-2:
       block-2;
       break;
…….
……..
default :
     default block;
     break;
……
}
Statement-x;
```

WAP to enter number and print corresponding day of week using switch….case

```
import java.io.*;
class day
  {
  public static void main(String args[]) throws IOException
   {
    int num;
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
  System.out.println("enter number :");
  num=Integer.parseInt(br.readLine());
switch(num)
{
case 1:
   System.out.println(" Monday");
   break;
case 2:
   System.out.println(" Tuesday");
   break;
case 3:
   System.out.println(" Wednesday");
   break;
case 4:
   System.out.println(" Thrusday");
   break;
```

```
case 5:
  System.out.println("Friday");
  break;
case 6:
  System.out.println(" Saturday");
  break;
case 7:
  System.out.println("Sunday");
  break;
default:
  System.out.println(" Please enter number between 1 to 7");
  break;
  }
 }
}
```

The ?: operator

The Java language has an unusual operator, useful for making two-way decisions. This operator is a combination of ? and : and takes three operands. This operator is popularly known as the conditional operator.

Syntax :

Conditional expression ? exp1:exp2

The conditional expression is evaluated first. If the result is nonzero, exp1 is evaluated. Otherwise exp2 is evaluated.

ex : if(x<0) flag=0;
flag=(x<0)? 0:1;
else
flag=1;

Switch case and nested ifs	**Nested if**
a) The switch() can only test for equality relational i.e only constant values are applicable.	a) The if can evaluate or logical expressions.
b) No two case statements have identical constants in the same switch.	b) Same conditions may be repeated for number of times.
c) Character constants are automatically converted to integers.	c) Character constants are automatically converted to integers.
d) If switch() case statement nested if can be used.	d) In nested if statement switch() case can be used.

Loop Control Statements

Loop :- A loop is defined as a block of statements which are repeatedly executed for certain number of times.

Java language provides for three constructs for performing loop operations :-

1) The while statement
2) The do statement
3) The for statement

The while statement

The simplest of all the looping structures in c is the while statement.

Syntax :-

```
initialization;
while (test condition)
  {
  body of the loop;
  increment/decrement;
  }
```

WAP to print 1…10 using while loop

```
class print
 {
 public static void main(String args[])
  {
   int i=1;
   while(i<=10)
     {
     System.out.println(i);
     i++;
     }
  }
 }
```

The for statement

The for loop is an entry-controlled loop that provides a more concise loop control structure.

Syntax :-

```
for (initialization;test-condition;incr/decr)
  {
  body of the loop;
  }
```

```
class loop1
{
public static void main(String args[])
 {
 int i;
```

```
 for(i=1;i<=10;i++)
   {
    System.out.println(i);
   }
  }
  }
```

The do statement

```
initialization;
do
  {
 body of the loop;
 incr/decr;
}while(condition);

class loop2
{
public static void main(String args[])
 {
 int i=1;
do
 {
 System.out.println(i);
 i ++;
}while(i<=10);
```

The for loop can be specified by different ways

1) for(;;) -> infinite loop -> No arguments
2) for(a=0;a<=20;) -> infinite loop -> 'a' is neither increased nor decreased
3) for(a=0;a<=10;a++) ->Display 1 to 10 -> 'a' is increased from 0 to 10.
   System.out.println(a);
4) for(a=10;a>=0;a--) -> Displays value -> 'a' is decreased from 10 to 0
   System.out.println(a); from 10 to 0

Note : The initialization section has two parts p=1 and n=1 separated by a comma.

Ex-
```
for(n=1,m=50;n<=m;n=n+1,m=m-1)
        {
        p=m/n;
        System.out.println(n,m,p);
        }
```
The multiple arguments in the increment section are separated by commas.

Note : It is also the test-condition may have any compound relation and the testing need not be limited only to the loop control variables.

e.g,
```
sum=0;
for(i=1;i<20  && sum<100; ++i)
```

```
{
sum=sum+i;
System.out.println(i+" " +sum);
}
```

WAP to enter no. and calculate the factorial using for loop.

```
import java.io.*;
class fact
  {
  public static void main(String args[]) throws IOException
 {
  int num,i,fact=1;
  System.out.println("Enter the number :");
  num=Integer.parseInt(br.readLine());
  for(i=1;i<=num;i++)
  {
  fact=fact*i;
  }
System.out.println("Factorial=" +fact);
}
}
```

1
12
123
1234
12345

```
class loop5
  {
  public static void main(String args[])
  {
  int i,j;
  for(i=1;i<=5;i++)
   {
   for(j=1;j<=i;j++)
     {
    System.out.print(j);
     }
  System.out.println();
 }
}
}
```

1
22
333
4444
55555

```
class loop6
   {
   public static void main(String args[])
  {
  int i,j;
  for(i=1;i<=5;i++)
   {
   for(j=1;j<=i;j++)
    {
    System.out.print(i);
    }
 System.out.println();
 }
}
}
```

12345
1234
123
12
1

```
class loop7
   {
   public static void main(String args[])
  {
  int i,j;
  for(i=5;i>=1;i--)
   {
   for(j=1;j<=i;j++)
    {
    System.out.print(j);
    }
 System.out.println();
 }
}
}
```

1
12
123
1234
12345

```
class loop8
   {
   public static void main(String args[])
  {
  int i,j,k,l=1;
```

```
  for(i=5;i>=1;i--)
   {
   for(j=1;j<=i;j++)
    {
    System.out.print(“ ”);
    }
  for(k=1;k<=l;k++)
   {
   System.out.print(k);
  }
 l=l+1;
System.out.println();
}
}
}
```

```
         *
        ***
       *****
     *******
   *********
```

```
class loop9
   {
   public static void main(String args[])
  {
  int i,j,k,l=1;
  for(i=5;i>=1;i--)
   {
   for(j=1;j<=i;j++)
    {
    System.out.print(“ ”);
    }
  for(k=1;k<=l;k++)
   {
   System.out.print(“*”);
  }
 l=l+2;
System.out.println();
}
}
}
```

WAP to enter number and check no. is prime or not.

```
import java.io.*;
class prime
  {
 public static void main(String args[]) throws IOException
  {
   int num,i;
```

```
  BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
  System.out.println("enter the number :");
  num=Integer.parseInt(br.readLine());
  for(i=2;i<num;i++)
   {
   if(num%i==0)
   break;
   }
 if(i==num)
  System.out.println(" Number is prime");
 else
  System.out.println(" Number is not prime");
  }
 }
```

WAP to enter number and reverse its digit.

```
 import java.io.*;
 class rev
  {
  public static void main(String args[]) throws IOException
  {
   int num,rev;
  BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
  System.out.println("enter the number :");
  num=Integer.parseInt(br.readLine());
   while(num>0)
    {
    rev=num%10;
    num=num/10;
    System.out.println(rev);
    }
  }
 }
```

WAP to enter number and print sum of its digit

```
 import java.io.*;
 class revsum
  {
  public static void main(String args[]) throws IOException
  {
   int num,rev,sum=0;
  BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
  System.out.println("enter the number :");
  num=Integer.parseInt(br.readLine());
   while(num>0)
    {
    rev=num%10;
    sum=sum+rev;
    num=num/10;
    }
 System.out.println("sum="+sum);
```

```
    }
  }
```

WAP to enter number and check no. is armstrong or not.

```
import java.io.*;
class arm
    {
  public static void main(String args[])  throws IOException
    {
     int num,rev,sum=0,temp;
     BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
     System.out.println("Enter the number :");
     Num=Integer.parseInt(br.readLine());
     temp=num;
     while(num>0)
       {
       rev=num%10;
       sum=sum+(rev*rev*rev);
       num=num/10;
       }
     if(temp==sum)
          System.out.println("Armstrong number");
     else
          System.out.println("Not armstrong number");
     }
   }
```

WAP to enter number and check no. is perfect or not.

```
import java.io.*;
class perfect
    {
  public static void main(String args[])  throws IOException
    {
     int num,i,sum=0,temp;
     BufferedReader br=new BufferedReader (new InputStreamReader (System.in));
     System.out.println("Enter the number :");
     num=Integer.parseInt(br.readLine());
     temp=num;
     for(i=1;i<=num/2;i++)
        {
         if(num%i==0)
            sum=sum+i;
        }
  if(temp==sum)
       System.out.println("Perfect number");
  else
       System.out.println("Not perfect number");
}}
```

Arrays -> An array is a group of contiguous or related data items that share a common name. For example, salary[10] represents the salary of 10^{th} employee.

One-dimensional array

A list of items can be given one variable name using only one subscript and such a variable is called a single- subscripted variable or a one-dimensional array.

For ex,

```
int number[] = new int[5];
```

```
class array1
 {
 public static void main(String args[])
   {
   String name[]={"Rohit","Amit","Niraj"};
   int roll[]={1,2,3};
   int age[]={25,26,27};
   int i;
   for(i=0;i<3;i++)
     {
     System.out.println("name="+name[i]);
     System.out.println("roll="+roll[i]);
     System.out.println("age="+age[i]);
     }
   }
 }
```

WAP to enter no. and find largest

```
import java.io.*;
class larger
  {
  public static void main(String args[]) throws IOException
    {
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
    int n,i,large;
    int num[]=new int[20];
    System.out.println("How many no. u want to enter :");
    n=Integer.parseInt(br.readLine());
    for(i=0;i<n;i++)
      {
      System.out.println("Enter number :");
      num[i]=Integer.parseInt(br.readLine());
      }
    large=num[0];
    for(i=0;i<n;i++)
      {
      if(large<num[i])
```

```
      large=num[i];
     }
   System.out.println("large=" +large);
   }
  }
```

WAP to enter number and find smallest

```
import java.io.*;
class smaller
 {
 public static void main(String args[]) throws IOException
  {
  BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
  int n,i,small;
  int num[]=new int[20];
  System.out.println("How many no. u want to enter :");
  n=Integer.parseInt(br.readLine());
  for(i=0;i<n;i++)
    {
    System.out.println("Enter number :");
    num[i]=Integer.parseInt(br.readLine());
    }
  small=num[0];
  for(i=0;i<n;i++)
    {
    if(small>num[i])
      small=num[i];
    }
  System.out.println("smaller=" +small);
  }
 }
```

Addition of two matrix

```
import java.io.*;
class sumofmatrix
 {
 public static void main(String args[]) throws IOException
  {
  BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
  int mat1[][]=new int[3][3];
  int mat2[][]=new int[3][3];
  int add[][]=new int[3][3];
  int i,j;
  System.out.println("Enter the value of first matrix :");
  for(i=0;i<3;i++)
    {
    for(j=0;j<3;j++)
     {
```

```
      mat1[i][j]=Integer.parseInt(br.readLine());
      }
     }
   System.out.println("Enter the value of first matrix :");
   for(i=0;i<3;i++)
    {
    for(j=0;j<3;j++)
     {
     mat2[i][j]=Integer.parseInt(br.readLine());
     }
    }
   for(i=0;i<3;i++)
    {
    for(j=0;j<3;j++)
     {
     add[i][j]=mat1[i][j]+mat2[i][j];
     }
    }
   for(i=0;i<3;i++)
    {
    for(j=0;j<3;j++)
     {
     System.out.print("\t" +add[i][j]);
     }
     System.out.println();
    }
   }
}
```

Class -> A class is a way of binding data and methods into a single unit. It provides a convenient method for packing together a group of logically related data items.

Object-> Instance of class is known as objects.

Syntax :

```
class class_name extends super_class_name
   {
   fields declaration;
   methods declaration;
   }
```

Here class_name and super_class_name are any valid java identifiers. The keyword extends indicates that the properties of the super_class_name class are extend to the class_name class.

Fields declaration

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called instance variables because they are created whenever an object of the class is instantiated.

```
class Rectangle
  {
  int length;
  int width;
  }
```

Methods declaration

It is a self-contained block of structure. Whatever method we write in the class are known as member methods. Whatever methods which does not come under scope of the class are known as non member methods. In java there are no non-member methods.

Syntax :

```
type method_name(parameter-list)
  {
  methods body;
  }
```

creating objects

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

Objects is java are created using the new operator. The new operator creates an object of the specified class and returns a reference to that object.

```
Rectangle rect1;                //declare the object
rect1=new Rectangle();          // instantiate the object
```

The first statement declares a variable to hold the object reference and the second one actually assigns the object reference to the variable. The variable rect1 is now an object of the rectangle.

Both statement can be combined into as :

```
Rectangle rect1=new Rectangle();
```

The method Rectangle() is the default constructor of the class. We can create any number of objects of rectangle.

Example :

```
Rectangle rect1=new Rectangle();
Rectangle rect2=new Rectangle();
```

And so on.

Note1 : Each object has its own copy of the instance variables of its class. This means that any changes to the variables of one object have no effect on the variables of another. It is also possible to create two or more references to the same object.

```
Rectangle r1=new Rectangle();
Rectangle r2=r1;
```

Accessing class members

We can't access the instance variables and the methods directly. To do this, we must use the concerned object and the dot operator.

```
object_name.variablename=value;
```

object_name.methodname(parameter-list);

WAP to print name, roll and age of a student

```
class student
 {
 String name;
 int roll,age;
 void getdata()
  {
  name="Amit kumar";
  roll=1;
  age=25;
  }
 void display()
  {
  System.out.println("name=" +name);
  System.out.println("roll=" +roll);
  System.out.println("age=" +age);
  }
public static void main(String args[])
  {
  student st=new student();
  st.getdata();
  st.display();
  }
}
```

Constructors in java

A constructor is a special member function which will be called by the JVM implicity for initializing the object whenever it is created.

Rules for writing the constructor

1) The name of the constructor must be same as the name of the class.
2) Constructors will be called automatically whenever an object is created.
3) Constructor should not return any value even void also if it is returning any value which will be treated as normal methods.

Once the constructor is private we can't create an object of that class into other class but it is possible to create the object of the same class.

Constructor should not be static.

Types of Constructor

Based on the uses of the constructors we have two types of constructors they are :

1) default or parameterless constructor
2) parameterized constructor

1) **Default constructor ->** A constructor is said to be default constructor if it is no parameter. It is also called parameterless constructor.

```
class student
  {
  string name;
  int roll,age;
  student()
    {
    name=”rahul”;
    roll=1;
    age=25;
    }
 void display()
   {
   System.out.println(“name=”+name);
   System.out.println(“roll=”+roll);
   System.out.println(“age=”+age);
   }
public static void main(String args[])
  {
  student st=new student();
  st.display();
  }
 }
```

Parameterize Constructor

A constructor is said to be a parameterize constructor if and only if it contains some parameters.

```
class pcons
  {
  String name;
  int roll,age;
    pcons(String nm,int r,int a)
    {
    name=nm;
    roll=r;
    age=a;
    }
  void display()
   {
   System.out.println("name=" +name);
   System.out.println("roll=" +roll);
   System.out.println("age=" +age);
   }
  }
 class pconsdemo
  {
```

```
 public static void main(String args[])
   {
   pcons p=new pcons("Amit",1,26);
   p.display();
   }
 }
```

Method overloading

Methods overloading that have the same name, but different parameter lists and different definitions. This is called method overloading. Method overloading is used when objects are required to perform similar tasks but using different input parameters. 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

```
class method
  {
  void add(int a,int b)
    {
    int sum=a+b;
    System.out.println("sum=" + sum);
   }
  void add(float c,float d)
   {
   float sum=c+d;
   System.out.println("sum=" + sum);
   }
  void add(double e,double f)
   {
   double sum=e+f;
   System.out.println("sum=" + sum);
   }
  void add(double g,double h,double i)
   {
  double sum=g+h+i;
  System.out.println("sum=" + sum);
   }
  }
class methodover
  {
 public static void main(String args[])
   {
  method m1=new method();
  m1.add(9,8);
  m1.add(8.4f,9.3f);
  m1.add(12.4,34.5);
  m1.add(12.3,23.3,12.4);
```

```
  }
 }
```

Overloaded constructor

A constructor is said to be overloaded constructor if and only if constructor name is similar but its parameters(signature) is different.

Signature is nothing but

a) type of parameters
b) No. of parameters
c) Order of parameters

Rules for overloading the constructor

1) constructor name must be similar.
2) Its signature must be different (either any one of the following three parameters must be different.

```
class cons
 {
 cons(int a,int b)
  {
  int sum=a+b;
  System.out.println("sum=" + sum);
 }
 cons(float c,float d)
  {
  float sum=c+d;
  System.out.println("sum=" + sum);
  }
 cons(double e,double f)
  {
  double sum=e+f;
  System.out.println("sum=" + sum);
  }
 cons(double g,double h,double i)
  {
 double sum=g+h+i;
 System.out.println("sum=" + sum);
  }
 }
class consdemo
 {
public static void main(String args[])
  {
 cons c1=new cons(10,12);
 cons c2=new cons(12.5f,6.5f);
 cons c3=new cons(12.4,34.5);
 cons c4=new cons(12.3,34.3,8.4);
 }
}
```

Data members

There are two types of data members in java-

i) Instance data members or object-level data members.
j) Static data members or class-level data members.

Methods

In java, we have two types of methods they are-

i) instance methods or object-level methods.
ii) static methods or class-level methods.

1) An IM is one which will be called each and every time with respect to an object.
   Objname.instancemethodname static methods are those will be called only with respect to class.
   Syntax : classname.staticmethodname

2) Instance methods are not sharable whereas static methods are sharable.
3) IMS are also known as object level methods whereas static methods are also known as class level methods.
4) Rtype mname(parameters)
   {
   block of statement;
   }

   static rtype mname(parameters if any)
   {
   block of statements;
   }

Static methods have several restrictions :

1) They can only call other static methods.
2) They can only access static data.
3) They can't refer to this or super in any way.

```
class MathOperation
   {
   static float mul(float x,float y)
      {
       return(x*y);
       }
    static float div(float x,float y)
       {
       return(x/y);
       }
   }
class MathApp
  {
  public static void main(String args[])
   {
   float a=MathOperation.mul(4.0f,5.0f);
   float b=MathOperation.div(a,2.0f);
   System.out.println("b=" +b);
  }
}
```

Inheritance

It's the mechanism of acquiring the property from one class to another class.

The class which is giving the properties and behaviour is known as super or base class or parent class.

ii) The class which is taking the properties is known as sub class or derived class or child class.

Note : The process of inheritance is also known as subclassing or reusability or extendable classes.

Subclass/Derived class -> A class is said to be a derived class if and only if it contains some features its own plus it inherits some properties from super class.

Advantages

i) ADT is reduced.
ii) Redundancy of the code is eliminated, hence it increases the inconsistencies.
iii) It reduces higher storage cost.
iv) Investment cost towards the project is reduced.

```
class <subclassname> extends <superclass>
   {
   List of properties;
   List of methods;
   }
```

Here subclassname and superclass represents the name of sub class and super class respectively.

Forms/ Types of inheritance

i) **single inheritance ->** An inheritance is said to be a single inheritance if and only if there will be a single super class and single sub class.

```
class A
  {
  ……
 …….
 }
class B extends A
 {
 …….
 …….
 }
```

// Single Inheritance

```
class A
{
```

```
    String name;
    int roll,age;
     void getdata()
    {
            name="Rohit";
            roll=1;
            age=25;
    }
     void display()
    {
            System.out.println("Name :" +name);
            System.out.println("Roll :"+roll);
            System.out.println("Age :"+age);
    }
}
class B extends A
{
    int m1,m2;
     void getm()
    {
            getdata();
            m1=80;
            m2=90;
    }
     void dispm()
    {
            display();
            System.out.println("M1="+m1);
            System.out.println("M2="+m2);
    }
};
class inheritdemo
{
    public static void main(String[] args)
    {
            B obj=new B();
            B.getm();
            B.dispm();
    }
}
```

ii) Multilevel inheritance -> It is one in which there will be a single super class, single sub class and n no. of intermediate super class.

```
Class A
  {
  …….
  …….
  }
Class B extends A
```

```
  {
  ……..
  ……..
  }
Class C extends B
  {
  ……
  ……
  }
```

Example :

```
class student
  {
  String name;
  int roll,age;
  void getdata()
    {
    name="Rohit";
    roll=5;
    age=25;
    }
   void putdata()
     {
     System.out.println("name=" +name);
     System.out.println("roll=" +roll);
     System.out.println("age=" +age);
     }
  }
class marks extends student
  {
  int m1,m2;
  void getm()
    {
    m1=75;
    m2=92;
    }
  void putm()
    {
    System.out.println("m1=" +m1);
    System.out.println("m2=" +m2);
    }
   }
 class sports extends marks
   {
   int spwt,total;
  void putwt()
    {
    spwt=80;
    putdata();
```

```
    putm();
    System.out.println("spwt=" +spwt);
    total=m1+m2+spwt;
    System.out.println("Total=" +total);
    }
  }
 class minherit
   {
   public static void main(String args[])
     {
     sports s = new sports();
     s.getdata();
     s.getm();
     s.putwt();
     }
  }
```

Method overriding

When a method in a sub class has the same name and type signature as a method in its super class then the method in sub class 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 :

```
class A
  {
  int i,j;
  A(int a,int b)
   {
   i=a;
   j=b;
   }
  void show()
   {
   System.out.println("i and j=" + i+" " +j);
   }
  }
class B extends A
  {
  int k;
  B(int a,int b,int c)
   {
   super(a,b);
   k=c;
   }
  void show()
   {
   super.show(); // super prevents override
```

```
        System.out.println("k=" +k);
       }
     }
     class override
       {
       public static void main(String args[])
        {
        B subob=new B(1,2,3);
        subob.show();
        }
      }
```

Abstract method

An abstract method represents undefined methods also known as unimplemented methods.

eg, abstract void getdata();

Abstract class

A class is said to be abstract class iff it contains at least one abstract method.

Or

A class is said to be abstract class iff it contains some concrete methods and some abstract methods. An abstract class is a class that can only be subclassed. It can't be instantiated.

Concrete Methods : A concrete methods represent defined methods.

eg,

```
abstract class Area
     {
      abstract void area();
     }
class Rectangle extends Area
  {
  public void area()
    {
    int length=10;
    int breadth=20;
    System.out.println("Area of rectangle =" + (length*breadth));
    }
 }
class Circle extends Area
  {
  public void area()
    {
    int radius=20;
    System.out.println("Area of circle is :" + 3.14*radius*radius);
    }
 }
class AbstDemo
```

```
  {
  public static void main(String args[])
   {
   Rectangle r=new Rectangle();
   r.area();
   Circle c=new Circle();
 c.area();
}
}
```

Interface

An interface is a collection of abstract methods and static final data members that can be implemented in by the any number of classes residing in a class hierarchy. Interface is known as a prototype for a class. Methods defined in an interface are only abstract methods.

Syntax :

```
interface <interface_name>
     {
      ……
     static final data member;
     return_type public methods(parameter);
     }

class <class_name> extends [super_class_name] implements [interface name]
     {
      …..
      ….
     }
```

Note : with respect to interface we can't instantiate a object directly.

eg,

```
class student
  {
  String stu_name;
  float fees;
  void getdata()
    {
    stu_name="Amit kumar";
    fees=1200.56f;
    }
  void display()
   {
  System.out.println("Name :" +stu_name);
  System.out.println("Fees :" +fees);
  }
}
class marks extends student
  {
```

```
  int m1,m2;
  void getm()
    {
    m1=20;
    m2=30;
    }
   void putm()
    {
    System.out.println("m1=" +m1);
    System.out.println("m2=" +m2);
    }
  }
interface sports
  {
  int spwt=30;
  void putsp();
  }
class result extends marks implements sports
   {
   int total;
   public void putsp()
    {
    total=m1+m2+spwt;
    display();
    putm();
    System.out.println("result=" +total);
   }
}
class face1
  {
public static void main(String args[])
  {
result r=new result();
r.getdata();
r.getm();
r.putsp();
}
}
```

Constants in java

A constant is the identifier which can't be change during execution of the program.

Note : To declare the variable such constants in java use a keyword called final.
final is a keyword which is acting as three places. They are :
1. variable level
2. method level
3. class level

1) At variable level -> If the final variable is initialized further modification are not allowed.

eg, final int a=10;
a=a+10; // not allowed

when the final variable is declared immediate assignment is possible but further modification are not possible.

e.g, final int a;
a=10; //allowed
a=a+10; // not allowed

2) At method level -> once the method is final which is not possible to override, i.e final methods can't be overridden in the sub classes.

Syntax : final rtype mname(mparams if any)
{
Block of statement;
}

Eg, final void sum() // not possible to override
{
……
}

3) At class level -> Once the class is final which can't be inheritable i.e final classes can't be inheritable or reusable.

```
final class a
  {
  ……..
  }
class b extends a        // error
   {
   ……
    }
```

Packages

A package is a collection of different types of class, interfaces and subpackages and in terms in subpackages is divided to classes, interfaces and subpackages and so on.

Advantage

1) To packages we can achieve the slogan of write once and reuse any number of times.

2) Application development time is reduced.
3) Replication of the code is reduced hence it increases the consistency of the application and reduces to storage cost.

There are two types of packages

1) pre – defined package
2) user defined package

1) **pre – defined package :-** A predefined package is one which comes along with the software to make the user applications in an efficient way to develop.

As the part of J2SE we have the following package :

i) **java.lang.* ->** used for specifying language functionality.
This is the default package will be imported our application automatically.
ii) **java.awt.* ->** used for developing GUI – oriented components (Radio Button, check box).
iii) **Java.io.* ->** used for developing string oriented application or file- oriented application.
iv) **Java.awt.event.* ->** used for developing event driven application for GUI component.
v) **Import java.applet.* ->** used for developing browser oriented application.
vi) **Import java.util.* ->** used for developing utility application (for achieving the performance in java/j2ee kind of projects. (collection framework).
vii) **import java.net.* ->** used for developing network related applications.

User – defined packages

A package is said to be user defined package if and only if which is developed by the user to place the list of classes, list of interface and sub-packages to simplicity the user application.

D:\> javac –d . <class_name>.java

creating packages :

To creating our own packages involves the following steps

1) Develop the package at the beginning of a file using the form

package <pkgname>;

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.

Note 1 :- java also supports the concept of package hierarchy. This is done by specifying multiple names in a package statement, separated by dots.

package first_package.second_package;

Accessing a package

Java package can be accessed either using a fully qualified class name or using a shortcut approach through the import statement. We use import statement when there are many references to a particular package or the package name is too long and unwidely.

The import statement can be used to search a list of packages for a particular class.

import package1.package2.classname;

We can have any no. of packages in a package hierarchy.

Note :- The statement must end with a semicolon (;) . The import statement should appear before any class definitions in a source file. Multiple import statements are allowed. eg, import firstpackage.secondpackage.myclass;

Note : After defining this statement, all the members of the class myclass can be directly access using the classname or its objects directly without using the package name.

We can also use another approach.

Syntax : import packagename.*;

The star(*) indicates that the compiler should search this entire package hierarchy when it encounters a class name. This implies that we can access all classes contained in the above package directly.

Advantages

The advantage is that we need not have to use long package repeatedly in the program.

Drawback

The major drawback of the shortcut approach is that it is difficult to determine from which package a particular member came.

```
package bank;
public class account
  {
  String name;
  int id,age;
  public void getdata(String na, int ag, int i)
    {
    name=na;
    id = i;
   age = ag;
    }
public void putdata()
  {
```

```
   System.out.println(“ name =” + name);
   System.out.println(“id =” +id);
   System.out.println(“ age=” +age);
  }
 }
 import bank.account;
 class bankdemo
  {
  public static void main(String args[])
    {
    account ac = new account();
    ac.getdata(“Amit kumar”,12,25);
    ac.display();
  }
}
```

1. What is polymorphism? Explain the difference between runtime and compile-time polymorphism?

Ans : polymorphism is a term that describes a situation where one name may refer to different methods. In java there are two types of polymorphism.

a) Compile-time polymorphism
b) Run-time polymorphism

a) Compile-time polymorphism -> It means function call is resolved at compile time. It means which function code will execute in response to a particular function call is known to compiler at compile time.
Example : method overloading.

b) Run-time polymorphism-> It means function call is resolved at run-time. It means which function code will execute in response to a particular function call is not known till execution of program.

Example : Dynamic method dispatch(DMD)

```
class A
 {
 void callme()
  {
  System.out.println("Inside A's callme method");
  }
 }
class B extends A
 {
 void callme()
  {
  System.out.println("Inside B's callme method");
  }
 }
class C extends A
```

```
 {
 void callme()
  {
  System.out.println("Inside C's callme method");
  }
 }
class dmd
 {
 public static void main(String args[])
  {
  A a=new A();
  B b=new B();
  C c=new C();
  A r;
  r=a;
  r.callme();
  r=b;
  r.callme();
  r=c;
  r.callme();
  }
 }
```

Access specifiers in Java

Java allows you to control access to classes, methods and fields so, called access specifiers. Java offers four access specifier.

i) public -> public classes, methods, and fields can be accessed from everywhere.

```
public class square
  {
  public int x,y,size;
  }
```

ii) protected -> protected methods and fields can only be accessed within the same class to which the methods and fields belong, within its subclasses, and within classes of the same package, but not from anywhere else.

```
public class abcd
 {
 protected int x,y;
 }
```

iii) default(no specifier) -> If we don't set access to specific level, then such a class, method, or field will be accessible from inside the same package to which the class, method, or field belongs, but not from outside this package.

iv) private -> private methods and fields can only be accessed within the same class to which the methods and fields belong. Private methods and fields are not visible within subclasses and are not inherited by subclasses.

```
public class square
  {
   private double x,y;
  }
```

Finalizer methods -> java supports a concept called finalization, which is just opposite to initialization. It automatically frees up the memory resources used by the objects. But objects may hold other non-object resources such as file descriptors or window System fonts. The garbage collector can't free these resources. In order to free these resources we must use a finalizer methods.

The finalizer methods is simply finalize() and can be added to any class. Java calls that method whenever it is about to reclaim the space for that object. The finalize method should explicitly define the tasks to be performed.

```
public void finalize()
  {
  ……
  }
```

this -> this is an implicit object by the JVM which is used for two purposes :

1) To distinguish between the formal parameters and class data members therefore the class data members must be preceded by implicit object called this.

2) "this" is an implicit object which is used to refer the current class object.

```
class rectangle
  {
  int length,breadth;
  void show(int length, int breadth)
    {
    this.length=length;
    this.breadth=breadth;
    }
 int calculate()
  {
 return(length*breadth);
   }
}
public class example
 {
 public static void main(String args[])
  {
  rectangle r=new rectangle();
  r.show(5,6);
  int area=r.calculate();
  System.out.println("the area of a rectangle is" + area);
  }
```

```
 }
```

super -> super is used for pointing the super class instance.

```
class a
  {
  int k=10;
  }
class test extends a
  {
public void m()
   {
System.out.println(super.k);
  }
class demo
   {
  public static void main(String args[])
    {
    Test t=new Test();
    t.m();
   }
}
```

super() -> It is used for calling super class default constructor. To call the default constructor of super classes explicitly no need to specify super.

Accessing superclass members

If our method overrides one of its superclass's methods, we can invoke the overridden method through the use of the keyword super. We can also use super to refer to a hidden field.

```
public class superclass
   {
   public void m1()
     {
     System.out.println("Super class");
     }
  }
public class subclass extends superclass
  {
public void m1()
  {
  super.m1();
  System.out.println("sub class");
  }
}
class pdemo
  {
public static void main(String args[])
  {
```

```
 subclass s=new subclass();
 s.m1();
 }
}
```

Exception Handling

An exception is an event, which occurs during the execution of a program that dirupt the normal flow of the program's instructions.

There are two types of exception :

i) pre-defined or built-in exception -> These usually indicate programming bugs, such as logic errors or proper use of an API. For example, division by zero problem.

Some built-in exceptions are :

a) ClassNotFoundException
b) CloneNotSupportedException
c) InterruptedException
d) NoSuchFieldException
e) NoSuchMethodException
f) ArithmeticException
g) ArrayStoreException
h) NumberFormatException
i) NullPointerException

Exception handling keywords

1) try -> The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block.

Syntax

```
try
  {
  code
  }
```

2) catch -> The block of the statements which are to be executed when an exception occurs must be written in catch block.

Every try block should contain atleast one catch block.

If an exception occurs within the try block, that exception is handled by an exception handler associated with it. To associate an exception handler with a try block, we must put a catch block after it.

A single try block can also contain multiple catch blocks.

Syntax :

```
try
  {
  …….
  }
catch(ExceptionType name)
```

```
        {
        …..
         }
       catch(ExceptionType name)
        {
        ….
       }
```

3) finally-> The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs.

```
finally
  {
  code
   }
```

4) throw -> All methods use the throw statement to throw an exception. The throw statement requires a single argument : a throwable object. Throwable objects are instances of any subclass of the throwable class.

Syntax : throw somethrowableobject;

5) throws-> If a method is capable of causing an exception that it doesn't handle, it must specify the throws behaviour.

The throws cause lists the types of exceptions that a method might throw. This throws cause is :

```
return type mname(parameter list) throws ExceptionList
    {
    ……..
     }
```

Example :

```
import java.io.*;
class example
  {
   public static void main(String args[])
     {
      try
      {
       int a=5,b=0;
       double c=a/b;
       System.out.println(c);
       }
      catch(ArithmeticException ae)
        {
        System.out.println(ae);
        }
     }
  }
```

User-defined Exceptions

A programmer can also his own set of exceptions. The advantage of creating a user-defined exception class is as follows :

i) It can hold more information about the error condition than a standard class. This becomes specially more important when the error condition is more complicated.

Eg,

```
 import java.io.*;
 class RangeException extends RuntimeException
     {
      public void disp()
           {
            System.out.println("The given no. not in 1 to 10");
            }  }
 class example
      {
      public static void main(String args[])
          {
           try
            {
        int x;
        DataInputStream dis=new DataInputStream(System.in);
        System.out.println("Enter a no. ");
        x=Integer.parseInt(dis.readLine());
        if(x<1 || x>10)
                throw new RangeException();
        else
                System.out.println("number accepted");
        }
        catch(IOException ioe)
           {
            System.out.println(ioe);
            }
         catch(RangeException re)
           {
           re.disp();
           }
        finally
           {
           System.out.println("I am from finally block");
           }
}
}
```

Applet programming

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 appletviewer or any web browser that supports java.

Difference between applet and application

i) Applet don't use the main() method for initiating the execution of the code. When loaded automatically call certain methods of applet class to start and execute the applet code.

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

iii) Applets can't read from or write to the files in the local computer.

iv) Applets are restricted from using libraries from other languages such as C or C++.

The steps involved in developing and testing in applet are :

1. Building an applet code (.java file)
2. Creating an executable applet (.class file)
3. Designing a web page using HTML Tags.
4. preparing <APPLET> tag.
5. Incorporating <APPLET> tag into the web page.
6. Creating HTML file.
7. Testing the applet code.

Write a simple program in Applet to print "Hello Java"

```
import java.awt.*;
import java.applet.*;
public class app1 extends Applet
  {
  public void paint(Graphics g)
    {
    g.drawString("Hello java",40,60);
    }
  }
```

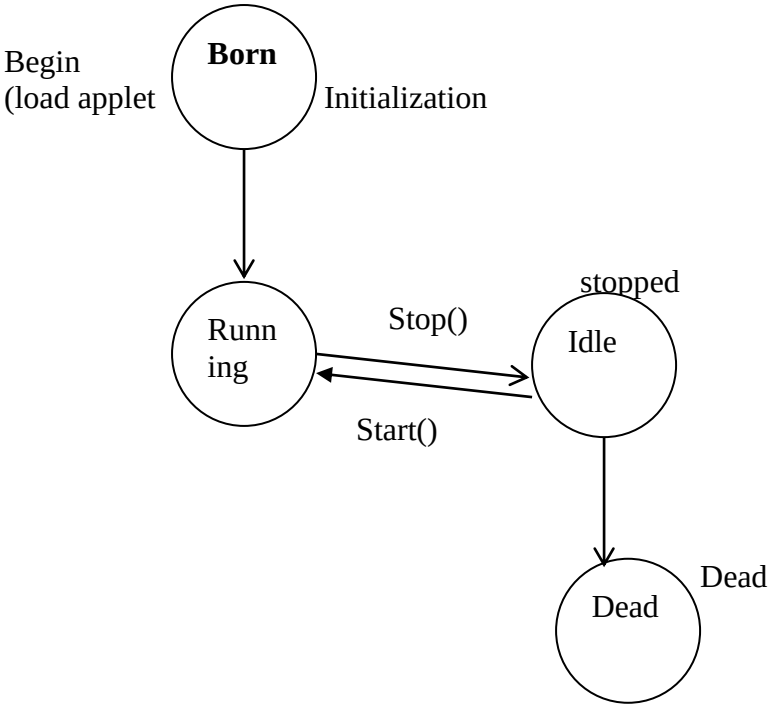
Compile -> javac app1.java

abc.html

```
<html>
<applet code="app1.class" width=400 height=400>
</applet>
</html>
```

For run
appletviewer abc.html

Applet life cycle



i) Initialization state -> Applet enters the initialization state when it is first loaded. This is achieved by calling the init() method of applet class. The applet is born. At this stage, we may do the following if required.
-> create objects needed by the applet
-> set up initial values.
-> load images or fonts.
-> setup colors.

```
public void init()
  {
  …….
  ……. (Action)
  }
```

ii) Running state :

Applet enters the running state when the system calls the start() method of applet class. This occurs automatically after the applet is initialized. Starting can also occur if the applet is already in stopped state.

```
public void start()
   {
   ……..
   …….
   }
```

iii) Idle or stopped state

An applet becomes idle when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet.

```
public void stop()
  {
  ……..
  …….
  }
```

iv) Dead state :-

An applet is said to be dead when it is removed from memory. This occurs automatically by invoking the destroy() method, when we quit the browser.

```
public void destroy()
 {
 ……..
 ……..
 }
```

v) Display state

```
 public void paint(Graphics g)
  {
  ……
  ……
  }
```

Displaying numeric values

In applets we can display numeric values by first converting them into strings and then using the drawstring() method of Graphics class. We can do this by calling the valueOf() method of string class.

```
import java.awt.*;
import java.applet.*;
public class numvalues extends Applet
 {
 public void paint(Graphics g)
  {
   int a=10;
   int b=20;
   int sum=a+b;
   String s="sum=" + String.valueOf(sum);
   g.drawString(s,100,100);
   }
 }
```

numvalues.html

```
<html>
<applet code=”numvalues.class” width=400 height=400>
</applet>
</html>
```

Getting input from the user

Applets work in graphical environment. Therefore, applets treat as text strings. We must first create an area of the screen in which user can type and edit input items.

Next step is to retrieve the items from the fields for display of calculations.

Develop an applet that receives three numeric values as input from the user and then displays the largest of the three on the screen. Write an HTML page and test the applet

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

public class large extends Applet
      {
      TextField text1,text2,text3;
      public void init()
         {
         text1 = new TextField(10);
         text2 = new TextField(10);
         text3 = new TextField(10);
         add(text1);
         add(text2);
         add(text3);
         }
      public void paint(Graphics g)
          {
           int num1 = 0;
           int num2 = 0;
           int num3 = 0;
           String s1, s2, s3;

g.drawString("Input a number in each box ", 10, 50);
try
  {
   s1 = text1.getText();
   num1 = Integer.parseInt(s1);
   s2 = text2.getText();
   num2 = Integer.parseInt(s2);
   s3= text3.getText();
   num3=Integer.parseInt(s3);
  }
catch(Exception e1)
   {}
```

```
    if(num1>num2)
          if(num1>num3)
             {
              String str="The largest is :" +String.valueOf(num1);
              g.drawString (str,100, 125);
              }
       else
           {
    String str1="The largest is :" +String.valueOf(num3);
    g.drawString (str1,100, 125);

    }
    else
         if(num2>num3)
              {
               String str3="The largest is :" +String.valueOf(num2);
               g.drawString (str3,100, 125);
               }
                 else
                   {
                    String str4="The largest is :"+String.valueOf(num3);
                   g.drawString (str4,100, 125);
                    }
          }
          public boolean action(Event ev, Object obj)
          {
                    repaint();
                    return true;
          }
}
```

WAP using Applet to calculate sum of its digit.

```
import java.applet.*;
import java.awt.*;
public class sum extends Applet
  {
   public void paint(Graphics g)
    {
    int num=123,rev,sum=0;
    while(num>0)
     {
     rev=num%10;
     sum=sum+rev;
     num=num/10;
     }
    String str="sum=" +String.valueOf(sum);
    g.drawString(str,100,25);
    }
  }
```

Coordinate System

By Default, the upper left corner of a GUI component (such as applet or window) has the coordinates (0,0). A Coordinate pair is composed of x-coordinate (the horizontal coordinate) and a y-coordinate (the vertical coordinate). The x-coordinate is the horizontal distance moving right from the upper left corner.

The y-coordinate is the vertical distance moving down from the upper left corner. The x-axis describes every horizontal coordinate, and the y-axis describes every vertical coordinate.

Drawing Lines

call
g.drawLine(x1,y1,x2,y2)
method, where `(x1, y1)` and `(x2, y2)` are the endpoints of our lines and g is the `Graphics` object we are drawing with. The following program will result in a line on the applet.

```
import java.applet.*;
import java.awt.*;
public class SimpleLine extends Applet
 {
  public void paint(Graphics g)
    {
     g.drawLine(10, 20, 30, 40);
     }
    }
```

Drawing Rectangles

Drawing rectangles is simple. Start with a `Graphics` object `g` and call its drawRect() method:

```
public void drawRect(int x, int y, int width, int height)
g.drawRoundRect(10,100,80,50,10,10);
```

The first argument `int` is the left hand side of the rectangle, the second is the top of the rectangle, the third is the width and the fourth is the height.

Drawing Ovals and Circles

Java has methods to draw outlined and filled ovals. These methods are called drawOval(), and fillOval() respectively. These two methods are:

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

Instead of dimensions of the oval itself, the dimensions of the smallest rectangle, which can enclose the oval, are specified.

If the width and height are same oval becomes a circle.

```
public void paint(Grpahics g)
  {
  g.drawOval(20,20,200,120);
  g.setColor(Color.green);
  g.fillOval(70,30,100,100); // This is a circle.
 }
```

Drawing Arcs

An arc is a part of an oval. The drawArc() designed to draw arcs takes six arguments. The first four are the same as the arguments for drawOval() method and the last two represent the starting angle of the arc and the number of degrees (sweep) angle around the arc.

```
g.drawArc(60,125,80,40,180,180);
```

```
import java.awt.*;
import java.applet.*;
public class line extends Applet
 {
 public void paint(Graphics g)
  {
  g.drawArc(50,50,100,50,180,180);
  g.drawOval(60,125,200,200);
  g.drawOval(60,125,200,100);
  g.drawRect(10,60,40,30);
  }
 }
```

USER INTERFACE COMPONENTS

The Button

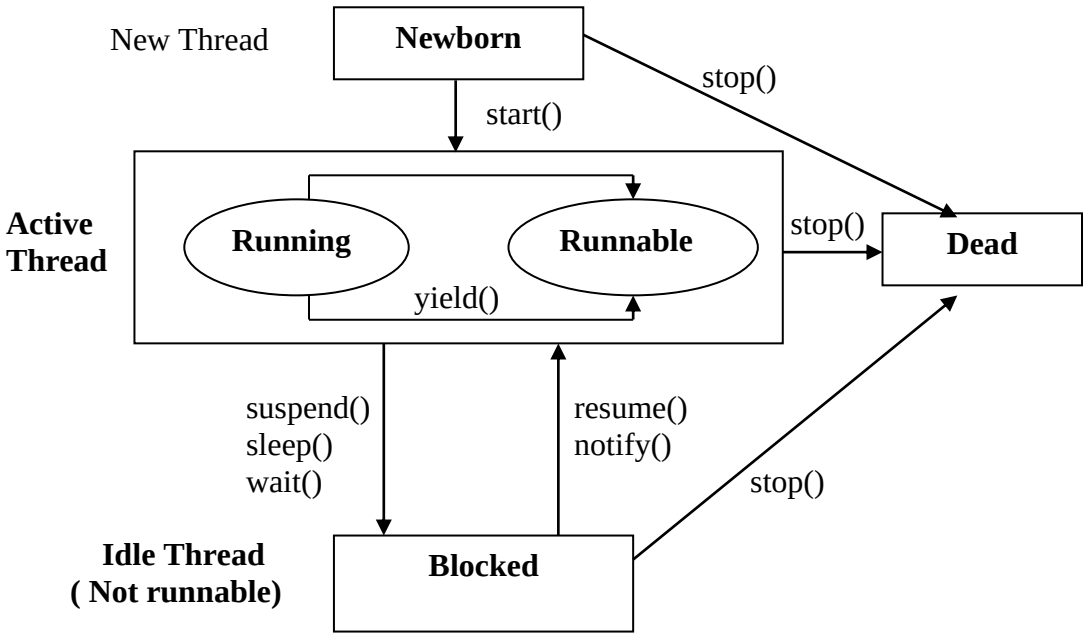
To create a button, use one of the following constructors:

Button() creates a button with no text label.
Button(String) creates a button with the given string as label.
Example:

```
import java.awt.*;
import java.applet.Applet;
public class ButtonTest extends Applet
   {
   Button b1 = new Button ("Play");
   Button b2 = new Button ("Stop");
   public void init()
     {
     add(b1);
     add(b2);
     }
   }
```

Q) Explain the states of a thread with figure.
Ans :

Life Cycle of a thread :



1) newborn state -> The thread is not yet scheduled for running.

2) runnable state -> The runnable state means that the thread is ready for execution and is waiting for the availability of the processor.

3) Running state -> It means that the processor has given its time to thread for its execution.

4) Blocking a thread -> A thread can also be temporarily suspended or blocked.

a) sleep() -> blocked for a specified time.
b) suspend() -> blocked until further orders.
c) wait() -> blocked until certain condition occurs.

The thread will return to the runnable state when the specified time is elapsed in the case of sleep() and the resume() method is invoked in case of suspend() and the notify() method is called in the case of wait().

5) Stopping a thread -> A thread move to the dead state.

A thread will also move to the dead state automatically when it reaches the end of its method.

Example :

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

```
   for(int i=1;i<=5;i++)
     {
     if(i==1) yield();
     System.out.println("i=" +i);
     }
   System.out.println("Exit from A");
  }
  }
class B extends Thread
  {
  public void run()
    {
    for(int j=1;j<=5;j++)
     {
     System.out.println("j=" +j);
     if(j==3) stop();
     }
   System.out.println("Exit from B");
  }
  }
class C extends Thread
  {
  public void run()
    {
    for(int k=1;k<=5;k++)
     {
     System.out.println("k=" +k);
     if(k==1)
     try
      {
      sleep(3000);
      }
     catch(Exception e)
     {}
     }
   System.out.println("Exit from C");
  }
  }
class thread
  {
  public static void main(String args[])
   {
   A a=new A();
   B b=new B();
   C c=new C();
   a.start();
   b.start();
   c.start();
   }
 }
```

Thread priority

Each thread is assigned a priority, which affects the order in which it is scheduled for running.

Java permits us to set the priority of a thread using the setPriority() method.

Threadname.setPriority(int number);

MIN_PRIORITY=1;

NORM_PRIORITY=5;

MAX_PRIORITY=10;

```
class A extends Thread
  {
  public void run()
    {
    for(int i=1;i<=5;i++)
      {
      System.out.println("i=" +i);
      }
    System.out.println("Exit from A");
    }
  }
class B extends Thread
  {
  public void run()
    {
    for(int j=1;j<=5;j++)
      {
      System.out.println("j=" +j);
      }
    System.out.println("Exit from B");
    }
  }
class C extends Thread
  {
  public void run()
    {
    for(int k=1;k<=5;k++)
      {
      System.out.println("k=" +k);
      }
    System.out.println("Exit from C");
    }
  }
class threada
  {
  public static void main(String args[])
    {
    A a=new A();
    B b=new B();
    C c=new C();
    a.setPriority(Thread.MIN_PRIORITY);
```

```
   a.start();
   b.setPriority(Thread.MAX_PRIORITY);
   b.start();
   c.setPriority(Thread.NORM_PRIORITY);
   c.start();
   }
 }
```

Synchronization

When one thread may try to read a record from a file while another is still writing to the same file then it may get strange results. Java enables us to overcome this problem using a technique known as synchronization.

```
synchronized void update()
  {
   // code here is synchronized
 }
```

When we declare a method synchronized, java creates a "monitor" hands it over to the thread that calls the thread holds monitor, no other thread can enter the synchronized section of code.

Whenever a thread has completed its work of using synchronized method, it will hand over the monitor to the next thread that is ready to use the same resource.

Implementing the Runnable interface

We can also create thread using runnable interface.

i) Declare the class as implementing the runnable interface.
ii) implement the run() method.
iii) create a thread by defining an object that is instantiated from this "runnable" class as the target of the thread.
iv) call the threads start() method to run the thread.

```
class X implements Runnable
  {
  public void run()
    {
    for(int i=1;i<=10;i++)
     {
     System.out.println("\tThread x :" + i);
     }
 System.out.println("end of threadX");
 }
}
class rtest
 {
 public static void main(String args[])
   {
   X r=new X();
   Thread tx=new Thread(r);
```

```
   tx.start();
   System.out.println("End of main thread");
  }
 }
```

Q) Explain the 'extends' keyword with an example.

Ans : 'extends' is a keyword where we can achieve the properties of inheritance.

Inheritance is the mechanism of obtaining the properties from one class to another class.

Note : The process of inheritance is also known as sub classing or extendable classes or reusability.

Example :

```
class student
  {
  String name;
  int roll,age;
  void getdata()
    {
    name="Rohit";
    roll=5;
    age=25;
    }
   void putdata()
     {
     System.out.println("name=" +name);
     System.out.println("roll=" +roll);
     System.out.println("age=" +age);
     }
  }
class marks extends student
  {
  int m1,m2;
  void getm()
    {
    m1=75;
    m2=92;
    }
  void putm()
    {
    System.out.println("m1=" +m1);
    System.out.println("m2=" +m2);
    }
   }
 class sports extends marks
   {
   int spwt,total;
  void putwt()
```

```
    {
    spwt=80;
    putdata();
    putm();
    System.out.println("spwt=" +spwt);
    total=m1+m2+spwt;
    System.out.println("Total=" +total);
    }
  }
 class inherit
   {
   public static void main(String args[])
     {
     sports s = new sports();
     s.getdata();
     s.getm();
     s.putwt();
     }
  }
```

Recursion

A method that calls itself is said to be recursive.

```
class fact
   {
   int fact(int n)
    {
     int result;
     if(n==1) return 1;
       result=fact(n-1) *n;
    return result;
   }
}
class r
   {
   public static void main(String args[])
     {
     fact f=new fact();
     System.out.println("Factorial of 3 is" + f.fact(3));
     System.out.println("Factorial of 4 is" +f.fact(4));
     System.out.println("Factorial of 5 is" +f.fact(5));
   }
}
```

Converting Strings to Numbers

When processing user input, it is often necessary to convert a String that the user enters into an int.

Syntax :

Integer.valueOf (Strings) and int Value () methods from the java.lang.Integer class. To convert the string "22" into the int .

int I = Integer.valueOf ("22").intValue();

Doubles, floats and long are converted similarly. To convert a String like "22" into the long value 22, we would write

long1 = Long.valueOf("22").longValue();

To convert "22.5" into a float or a double, we would write:
double x = Double.valueOf("22.5").doubleValue();
float y = Float.valueOf("22.5").floatValue();
The various **valueOf()** methods are relatively intelligent and can handle plus and minus signs, exponents, and most other common number formats. However, if we pass one of these methods, something completely non-numeric, like "hello world," it will throw a **NumberFormatException.**

class area
{
public static void main (String args[])
{
double radius:
double pi = 3.14;
double A
radius = Double.valueOf(args[0].doubleVaue ()
A = pi*radius*radius:
System.out.printin(A + "square meters")
}
}

WRAPPER CLASSES

In Java, primitive data types such as long, int, char etc. are not treated as objects. This is for simplicity and efficiency; but many times, we need to treat these primitive type as objects, for example, when we want to store primitive data types along with other objects in an array. The Java programming language provides **"Wrapper"** to manipulate primitive data elements as objects. Each primitive data type has a corresponding wrapper class in the java.lang package. These classes are:

Primitive Data Type	Wrapper class
Boolean	Boolean
Byte	Byte
Char	Character
Short	Short
Int	Integer
Long	Long
Float	Float

A wrapper class object is constructed by passing the value to be wrapped into the appropriate constructor. For example:

int ptyp=10;
Integer pobj = new Integer(ptyp);

INNER CLASSES

A class that is declared and defined inside some other class is called an **inner class.** Sometimes, it is also called a nested class. The inner classes give additional functionality to the program, and make it clearer.

```
public class Outer
{
int i=10;
public class Inner
{
int j=20;
public void innerFn()
{
System.out.printIn("j is "+ j);
}
}
public void outerFn()
{
System.out.printIn("I is "+I);
}
public static void main(String s [ ])
{
Outer out =new Outer();
out.outerFn();
}
}
```

Enclosing the 'this' reference of Inner Classes.

Inner classes have access to their enclosing class's scope. The access to enclosing class's scope is possible because the **inner class** actually has a hidden reference to the outer class **this reference.**

Example:

```
public class Outer
{
int i=10;
public class Inner
{
int j;
public void innerFn()
{
System.out.printIn("I is "+I);
System.out.printIn("j is "+j);
```

```
}
}
public void innerCreate()
{
Inner in=new Inner();
in.innerFn();
}
public void outerFn()
{
System.oaut.printIn("I is "+ i);
}
public static void main(String s[ ] )
{
Out out=new Outer();
out.innerCreate();
}
}
```

ADDING CLASSES TO EXISTING PACKAGES

Suppose a package A contains a public class, Class A, and we want to add another public class, Class B to this package.

```
package packageA;
public class ClassA;
{
//(body of classA)
}

package packageB;
public class ClassB
{
//(body of ClassB)
}
```

Store the source and compiled files ClassB.java and ClassB.class in the directory **packageA**. we can add non-public classes also in this manner. When the package, **packageA**, is imported, both the classes **ClassA** and **ClassB** are imported, as they are contained in that package.

A **.java** file can have only one public class. So, if we want to create a package with multiple public classes in it, create the classes in separate source files and declare the package statement

package packagename;

at the top of each source file. Switch to the subdirectory created with the package name, and compile each source file. Now, the package contains **.class** files of all the source files.

INTERTHREAD COMMUNICATION

To avoid wastage of precious time of CPU, or to avoid polling, Java includes an interprocess communication mechanism via the wait( ), notify( ), and notifyAll( ) methods. These methods are implemented as final methods, so all classes have them. These three methods can be called only from within a synchronized method.

wait( ): Tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll().
notify( ): Wakes up a single thread that is waiting on this object's monitor.
notifyAll( ): Wakes up all threads that are waiting on this object's monitor, the highest priority Thread will be run first.

These methods are declared within objects as following:

final void wait( ) throws Interrupted Exception
final void notify( )
final void notifyAll( )

These methods enable you to place threads in a waiting pool until resources become available to satisfy the Thread's request. A separate resource monitor or controller process can be used to notify waiting threads that they are able to execute.

//program

```
class WaitNotifyTest implements Runnable
{
WaitNotifyTest ()
{
Thread th = new Thread (this);
th.start();
}
synchronized void notifyThat ()
{
System.out.println ("Notify the threads waiting");
this.notify();
}
synchronized public void run()
{
try
{
System.out.println("Thead is waiting....");
this.wait ();
}
catch (InterruptedException e){}
System.out.println ("Waiting thread notified");
}
}
class runWaitNotify
{
public static void main (String args[])
```

```
{
WaitNotifyTest wait_not = new WaitNotifyTest();
Thread.yield ();
wait_not.notifyThat();
}
}
```

I/O in java

Java input and output are based on the use of streams, or sequences of bytes that travel from a source to a destination over a communication path. If a program is writing to a stream, you can consider it as a stream's source. If it is reading from a stream, it is the stream's destination. The communication path is dependent on the type of I/O being performed. It can consist of memory-to-memory transfers, a file system, a network, and other forms of I/O. Streams are powerful because they abstract the details of the communication path from input and output operations. This allows all I/O to be performed using a common set of methods. These methods can be extended to provide higher-level custom I/O capabilities.

Three streams given below are created automatically:
System.out - standard output stream
System.in - standard input stream
System.err - standard error.

An **InputStream** represents a stream of data from which data can be read. Again, this stream will be either directly connected to a device or else to another stream.

An **OutputStream** represents a stream to which data can be written. Typically, this stream will either be directly connected to a device, such as a file or a network connection, or to another output stream.

STREAMS AND STREAM CLASSES

There are two types of streams: byte streams, and character streams.

Byte streams carry integers with values that range from 0 to 255. A diversified data can be expressed in byte format, including numerical data, executable programs, and byte codes – the class file that runs a Java program.

Character Streams are specialised types of byte streams that can handle only textual data.

Byte Stream Classes

Java defines two major classes of byte streams: InputStream and OutputStream. To provide a variety of I/O capabilities subclasses are derived from these InputStream and OutputStream classes.

InputStream class

The InputStream class defines methods for reading bytes or arrays of bytes, marking locations in the stream, skipping bytes of input, finding out the number of bytes available for reading, and resetting the current position within the stream. An input stream is automatically opened when created. The close() method can explicitly close a stream.

Methods of InputStream class

The basic method for getting data from any InputStream object is the read() method.

public abstract int read() throws IOException: This reads a single byte from the input stream and returns it to the stream.

public int read(byte[] bytes) throws IOException: fills an array with bytes read from the stream and returns the number of bytes read.

public int read(byte[] bytes, int offset, int length) throws IOException: fills an array from a stream starting at position offset, up to the length of the bytes. This returns either the number of bytes read or 1 for end of file.

public int available() throws IOException: the read()method always blocks when there is no data available. To avoid blocking, the program might need to ask ahead of time exactly how many bytes it can safely read without blocking. This method returns this number.

public long skip(long n): the skip() method skips over n bytes (passed as argument of skip()method) in a stream.

OutputStream class

The OutputStream defines methods for writing bytes or arrays of bytes to the stream. An output stream is automatically opened when created. An Output stream can be explicitly closed with the close() method.

Methods of OutputStream class

public abstract void write(int b) throws IOException: writes a single byte of data to an output stream.
public void write(byte[] bytes) throws IOException: writes the entire contents of the bytes array to the output stream.
public void write(byte[] bytes, int offset, int length) throws IOException: writes length, number of bytes starting at position offset from the bytes array.

The Java.io package contains several subclasses of InputStream and OutputStream that implement specific input or output functions. Some of these classes are:

FileInputStream and FileOutputStream: read data from or write data to a file on the native file system.

Character Stream Classes

Character Streams are defined by using two class **Java.io.Reader and Java.io.Writer** hierarchies.

Both Reader and Writer are the abstract parent classes for character stream based classes in the Java.io package. Reader classes are used to read 16-bit character streams and Writer classes are used to write to 16-bit character streams. The methods for reading from and writing to streams found in these two classes and their descendant classes (which we will discuss in the next section of this unit) given below:

int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)

Reading Console Input

Java takes input from the console by reading from System.in. It can be associated with these sources with reader and sink objects by wrapping them in a reader object. For System.in an InputStreamReader is appropriate. This can be further wrapped in a BufferedReader as given below, if a line-based input is required.

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

After this statement br is a character-based stream that is linked to the console through Sytem.in

The program given below is used for receiving console input in any of your Java applications.

//program

```
import Java.io.*;
class ConsoleInput
{
static String readLine()
{
StringBuffer response = new StringBuffer();
try
{
BufferedInputStream buff = new BufferedInputStream(System.in);
int in = 0;
char inChar;
do
{
in = buff.read();
inChar = (char) in;
```

```
if (in != -1)
{
response.append(inChar);
}
} while ((in != 1) & (inChar != '\n'));
buff.close();
return response.toString();

}
catch (IOException e)
{
System.out.println("Exception: " + e.getMessage());
return null;
}
}
public static void main(String[] arguments)
{
System.out.print("\nWhat is your name?");
String input = ConsoleInput.readLine();
System.out.println("\nHello, " + input);
}
}
```

Writing Console Output

we have to use System.out for standard Output in Java. It is mostly used for tracing errors, or for sample programs. These sources can be associated with a writer and sink objects by wrapping them in writer object. For standard output, an OutputStreamWriter object can be used, but this is often used to retain the functionality of print and println methods. In this case, the appropriate writer is PrintWriter. The second argument to PrintWriter constructor requests that the output will be flushed whenever the println method is used. This avoids the need to write explicit calls to the flush method in order to cause the pending output to appear on the screen. PrintWriter is different from other input/output classes as it doesn't throw an IOException. It is necessary to send check Error message, which returns true if an error has occurred. One more side effect of this method is that it flushes the stream.

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

Object serialization

Serialization takes all the data attributes, writes them out as an object, and reads them back in as an object. For an object to be saved to a disk file it needs to be converted to a serial form. An object can be used with streams by implementing the serializable interface. The serialization is used to indicate that objects of that class can be saved and retrieved in serial form. Object serialization is quite useful when you need to use object persistence. By object persistence, the stored object continues to serve the purpose even when no Java program is running, and stored information can be retrieved in a program so it can resume functioning, unlike the other objects that cease to exist when object stops running.

DataOuputStreams and DataInputStreams are used to write each attribute out individually, and then can read them back in at the other end. But to deal with the entire object, not its individual attributes, store away an object or send it over a stream of objects.

Transient Keyword

When an object that can be serialized, we have to consider whether all the instance variables of the object will be saved or not. Sometimes, we have some objects or sub objects which carry sensitive information like a password. If we serialize such objects even if information (sensitive information) is private in that object it can be accessed from outside. To control this we can turn off serialization on a field- by-field basis using the transient keyword.

//Program

```
import Java.io.*;
import Java.util.*;
public class SerialDemo implements Serializable
{
private Date date = new Date();
private String username;
private transient String password;
SerialDemo(String name, String pwd)
{
username = name;
password = pwd;
}
public String toString()
{
String pwd = (password == null) ? "(n/a)" : password;
return "Logon info: \n " + "Username: " + username +
"\n Date: " + date + "\n Password: " + pwd;
}
public static void main(String[] args)
throws IOException, ClassNotFoundException
{
SerialDemo a = new SerialDemo("Java", "sun");
System.out.println( "Login is = " + a);
  ObjectOutputStream 0 = new ObjectOutputStream( new
  FileOutputStream("Login.out"));
0.writeObject(a);
0.close();
// Delay:
int seconds = 10;
long t = System.currentTimeMillis()+ seconds * 1000;
while(System.currentTimeMillis() < t)
;
// Now get them back:
ObjectInputStream in = new ObjectInputStream(new
FileInputStream("Login.out"));
```

```
System.out.println("Recovering object at " + new Date());
a = (SerialDemo)in.readObject();
System.out.println( "login a = " + a);
}

}
```

Volatile Modifier

The volatile modifier is used when we are working with multiple threads. The Java language allows threads that access shared variables to keep private working copies of the variables. This allows for a more efficient implementation of multiple threads. These working copies need to be reconciled with the master copies in the shared (main) memory only at prescribed synchronization points, namely when objects are locked or unlocked. As a rule, to ensure that shared variables are consistently and reliably updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock and conventionally enforcing mutual exclusion for those shared variables. Only variables may be volatile. Declaring them so indicates that such methods may be modified asynchronously.

USING NATIVE METHODS

Native Method is a method which is not written in Java, and is outside of the JVM in a library. This feature is not special to Java. Most languages provide some mechanism to call routines written in another language. In C++, you must use the extern "C" statement to signal that the C++ compiler is making a call to C functions.
To declare a native method in Java, a method is preceded with native modifiers much like we use the public or static modifiers, but don't define any body for the method, but simply place a semicolon in its place.

Syntax :

```
native void getdata();
```

Q. Differentiate between call-by-value and call-by-reference with the help of suitable example.

Ans : Both call by value and call by reference is used to pass the argument in a function.

Call by value-> This method copies the value of an argument into the format parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

```
class Test
  {
  void call(int i,int j)
    {
    i=i*2;
    j=j/2;
    }
  }
```

```
class callbyvalue
 {
 public static void main(String args[])
  {
  Test ob=new Test();
  int a=15;
  int b=20;
  System.out.println(" a and b before call" +a + " " +b);
  ob.call(a,b);
  System.out.println("a and b after call" +a + " " +b);
  }
 }
```

Call by reference -> In this method, a reference to an argument is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will effect the argument used to call the subroutine.

```
class Test
 {
 int a,b;
 test(int i,int j)
  {
  a=i;
  b=j;
  }
 void call(Test o)
  {
  o.a*=2;
  o.b/=2;
  }
 }
class callbyref
 {
 public static void main(String args[])
  {
  Test ob=new Test(15,20);
  System.out.println(" a and b before call" +ob.a + " " +ob.b);
  ob.call(ob);
  System.out.println("a and b after call" +ob.a + " " +ob.b);
  }
 }
```

Q. Two methods used for applet execution

Ans : The two methods for executing an applet in java are :

i) One using internet explorer and using HTML file.
ii) Second using class AppletViewer.

Q. Write a recursive function in java to print the gcd of two given integers.

Ans :

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

```
        {
           int a, b;
                 //User input for both bumbers
           a=Integer.parseInt(args[0]);
           b=Integer.parseInt(args[1]);

                 System.out.println("The gcd of "+a+" and "+b+" is "+gcd(a,b)+".");
        }
     static int gcd(int a,int b)
        {
           if(b==0) return a;
                 else return gcd(b,a%b);
        }
}
```

Q. Static function can only use static data of the class, justify your answer.

Ans : static variables is the variable which use to have static value for all the class. It is common for all the instance of the class. When an instance variable is declared static, then at the time of instantiation of class, it is initialize.

If one want to initialize all the static variables, then a member of static type can be created. Since the method is static, so one can access it without instantiation of class also.

e.g,

```
class xyz
      {
      static int n1=5;
      static int n2;
      static void init(int x)
        {
        System.out.println("n1=" +n1);
        System.out.println("n2=" +n2);
        System.out.println("x=" +x);
        }
     public static void main(String args[])
        {
        init(10);
        }
```

Q. What are the points to ensure while writing a method that is intended to override another method.

Ans : Some important points that must be taken care while overriding a method.
a) An overriding method replaces the method it overrides.
b) Each method in a parent class can be overrides at most once in any of the subclass.
c) Overriding methods must have exactly same arguments lists, both in type and in order.
d) an overriding method must have same return type as the method it overrides.
e) Overriding is associated with inheritance.

Q. Differentiate between interfaces and abstract classes with the help of suitable examples.

Ans : Abstract class and interfaces similarity between then that methods of both should be implemented but there are many differences between then these are :

i) A class can implement more than one interface, but abstract class can only subclass one class.
ii) An abstract class can have non-abstract methods. All methods of an interface are implicitly or explicitly abstract.
iii) An abstract class can declare instance variable but an interface can't.
iv) Every method of interface is simplicitly public.

Q. What is data hiding? Explain with example.

Ans : The wrapping of data and functions into a single unit is known as encapsulation and unit is known as class. The data is not directly accessible to the outside world and only in functions, which are wrapped in the class, can access it. Functions are made available to outside world. These functions provides an interface to access data. If one wants to modify data of an object. It should known exactly the functions that are available to interact with it. This insulation of data from direct access by program is known as data hiding. E.g,

```
class Area
  {
  private int a,b;
  public void calArea(int x,int y)
     {
     a=x;
     b=y;
     return a*b;
     }}
```

Q. Write a recursive function in java to print a given string in reverse order.

Ans :

```
import java.io.*;
class revorder
{
public static void main(String args[]) throws IOException
{
BufferedReader ob=new BufferedReader(new InputStreamReader(System.in));
System.out.println("type in a statement you want to reverse");
String st=ob.readLine();
String revstr="";
int len=st.length();
char sp=' ';
for(int i=len;i<=0;i++)
{
if(st.charAt(i)==sp)
{
for(int j=len;i<=0;j--)
{
revstr=revstr+st.charAt(i);
}}
System.out.println(revstr);}}}
```

1. What is java Platform?

Ans : When Java Code is compiled a byte code is generated which is independent of the system. This byte code is fed to the JVM (Java Virtual Machine) which resides in the system. Since every system has its own JVM, it doesn't matter where we compile the source code. The byte code generated by the compiler can be interpreted by any JVM of any machine. Hence it is called Platform independent Language.

Java's byte code are designed to be read and interpreted in exactly same manner on any computer hardware or operating system that supports Java Runtime Environment.

When we write the program for any programming language it is called (source program), and it will have the extension depending upon the language.

In Java, when we write the program, we will have the ".java" file. And when it is compiled, we will get the ".class" file. The ".class" file is executed by the Java Virtual Machine (JVM). If we have the JVM, we can execute the java program anywhere under operating system. That means, that Java is PLATFORM INDEPENDENT.

Q. What is a default layout manager in java. Explain it.

i) Border Layout

The border layout is the default layout manager for all Window objects. It consists of five fixed areas: North, South, East, West, and Center. You do not have to put a component in every area of the border layout. The demonstration puts the label "Border Layout" in the North area, the OK button in the South area, and the List of fruit trees in the Center area. The East (right) and West (left) areas are empty.

If any or all of the North, South, East, or West areas are left out, the Central area spreads into the missing area or areas. However, if the Central area is left out, the North, South, East, or West areas do not change.

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

Void add(Component compObj, Object region);

Here, compObj is the component to be added, and region specifies where the component will be added.

ii) FlowLayout

The flow layout is the default layout manager for all Panel objects and applets. It places the panel components in rows according to the width of the panel and the number and size of the components. 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.

1. Explain any six bitwise operators of Java. Also, demonstrate each operator with an example.

Ans : There are different operations can be performed using bitwise operators:

a) bitwise AND-> The bitwise AND operation will be carried out between the two bit patterns of the two operands. For example,

Truth Table of AND

A	B	F (Output)
0	0	0
0	1	0
1	0	0
1	1	1

Example

	Binary pattern
i) int x = 5;	0101
int y = 2;	0010
int z = x & y;	0000

so, z = 0

b) bitwise OR-> The bitwise OR operations are similar to the bitwise AND and the result is 1 if any one of the bit value is 1. The symbol | represents the bitwise OR.

Truth Table of OR

A	B	F (Output)
0	0	0
0	1	1
1	0	1
1	1	1

Example :

	Binary Pattern
i) int x = 5;	0101
int y = 2;	0010
int z = x | y	0111

so, z = 7

c) bitwise exclusive OR-> The bitwise exclusive OR will be carried out by the notation ^. To generate a 1 bit in the result, a bitwise exclusive OR needs a one in either number but not in both.

Truth Table of XOR

A	B	F (Output)
0	0	0
0	1	1
1	0	1
1	1	0

Example :

	Binary Pattern		**Binary Pattern**
i) int x = 5;	0101	**ii)** int a = 6	0110
int y = 2;	0010	int b = 3	0011
int z = x^y	0111	int c = a^b	0101
so, z = 7		so, c = 5	

d) Shift Operations -> The shift operations take binary patterns and shift the bits to the left or right, keeping the same number of bits by dropping shifted bits off the end and filling in with zeros from the other end. Java provides two types of shift operations, left shift and right shift.

i) left shift -> The << operator is used for left shifting.

Example :

Variable	**Value**	**Binary Patterns**
i) X	33	0010 0001(8 bits)
X<<1		- the bit pattern of the X value is left shifted once.
Output :		0100 0010
So, result is : 66		
ii) int X = 33		0010 0001(8 bits)
X<<3		the bit pattern of the X value is left shifted by thrice.
		0 0100 0010
		0 1000 0100
		1 0000 1000
Since, The resultant is 8 bit pattern		
So, the resultant bit pattern will be : 0000 1000		
So, result is : 8		

ii) right shift -> The right shift >> operator is used for right shifting.

Example :

	Binary Pattern
i) y = 41,	0010 1001
y>>3	the bit pattern of the y value is right shifted by thrice.
00101001	
00010100	1
00001010	0
00000101	0

So, the resultant bit pattern will be : 00000101

So, result is : 5

e) Bitwise Complement-> The complement operator ~ switches all the bits in a binary pattern, that is, all the zeroes become ones and all the ones become zeroes. The complement of a pattern is often useful in signaling and controlling other devices where several different signals may be complementary to each other.

Example :

Variable	**Value**	**Binary Pattern**
X	23	0001 0111(8 bits)
~X	132	1110 1000
Y	ff	1111 1111
~Y	00	0000 0000

Q. What is JVM. Explain

A **Java Virtual Machine** (**JVM**) enables a set of computer software programs and data structures to use a virtual machine model for the execution of other computer programs and scripts. The model used by a JVM accepts a form of computer intermediate language commonly referred to as Java bytecode.

A JVM can also implement programming languages other than Java. For example, Ada source code can be compiled to Java byte code, which may then be executed by a JVM. JVMs can also be released by other companies besides Oracle (the developer of Java) — JVMs using the "Java" trademark may be developed by other companies as long as they adhere to the JVM specification published by Oracle and to related contractual obligations.

Java was conceived with the concept of WORA: "write once, run anywhere". This is done using the Java Virtual Machine. The JVM is the environment in which java programs execute. It is software that is implemented on non-virtual hardware and on standard operating systems.

JVM is a crucial component of the Java platform, and because JVMs are available for many hardware and software platforms, Java can be both middleware and a platform in its own right, hennce the trademark write once, run anywhere. The use of the same byte code for all platforms allows Java to be described as "compile once, run anywhere", as opposed to "write once, compile anywhere", which describes cross-platform compiled languages.

Q. Write a java program using class and constructor to find the length of string.

```
import java.lang.reflect.*;

  public class DumpMethods {
    public static void main(String args[])
    {
      try {
        Class c = Class.forName(args[0]);
        Method m[] = c.getDeclaredMethods();
        for (int i = 0; i < m.length; i++)
        System.out.println(m[i].toString());
      }
      catch (Throwable e) {
        System.err.println(e);
      }
    }
  }
```

Q. Explain String and StringBuffer class. Explain various string functions.

Java provides the StringBuffer and String classes, and the String class is used to manipulate character strings that cannot be changed. Simply stated, objects of type String are read only and immutable. The StringBuffer class is used to represent characters that can be modified.

The significant performance difference between these two classes is that StringBuffer is faster than String when performing simple concatenations. In String manipulation code, character strings are routinely concatenated. Using the String class, concatenations are typically performed as follows:

```
String str = new String ("Stanford  ");
str += "Lost!!";
```

If we were to use StringBuffer to perform the same concatenation, we would need code that looks like this:

```
StringBuffer str = new StringBuffer ("Stanford ");
str.append("Lost!!");
```

Various string functions

1. public String concat(String s)

This method returns a string with the value of string passed in to the method appended to the end of String which used to invoke the method.

```
String s="abcdefg";
System.out.println(s.concat("hijlk"));
```

```
s.concat("hijkl");
System.out.println(s);
```

2. public charAt(int index)

This method returns a specific character located at the String's specific index.Remember,String indexes are zero based.Example

```
String s="Alfanso Mango";
System.out.println(s.charAt(0));
```

The output is 'A'

3.public int length()

This method returns the length of the String used to invoke the method.example:-

```
String s="name";
System.out.println(s.length());
```

The output is 4

4.public String replace(char old, char new)

This method return a String whose value is that of the String to invoke the method ,updated so that any occurrence of the char in the first argument is replaced by the char in the second argument .Example:-

```
String s="VaVavavav";
System.out.println(s.replace('v','V'));
```

The output is VaVaVaVaV

5.public boolean equalsIgnoreCase(String s)

This method returns a boolean value depending on whether the value of the string in the argument is the same as the String used to invoke the method.This method will return true even when character in the string object being compared have different cases.Example:-

```
String s="Vaibhav";
System.out.println(s.equalsIgnoreCase("VAIBHAV"));
```

The output is true

6.public String substring(int begin) / public String substring(int begin,int end)

substring method is used to return a part or substring of the String used to invoke the method.The first argument represents the starting location of the substring.Remember the indexes are zero based.example:-

```
String s="abcdefghi";
System.out.println(s.substring(5));
System.out.println(s.substring(5,8));
```

The output would be
" fghi "
" fg "

7.public String toLowerCase()

This method returns a string whose value is the String used to invoke the method,but with any uppercase converted to lowercase.:-

```
String s="AbcdefghiJ";
System.out.println(s.toLowerCase());
```

Output is " abcdefghij "

8.public String trim()

This method returns a String whose value is the String used to invoke the method ,but with any leading or trailing blank spaces removed.Example:-

```
String s="hey here is the blank space ";
System.out.println(s.trim())
```

The output is " heyhereistheblankspace"

Q. What is labeled break in java. Explain.

1. The break statement can be followed by a label.
2. The presence of a label will transfer control to the start of the code identified by the label.

```
public class Simple
 {
 public static void main(String[] args)
   {
    OuterLoop: for (int i = 2;i<=50; i++)
{
 for (int j = 2; j < i; j++) {
  if (i % j == 0) {
    continue OuterLoop;
  }
 }
 System.out.println(i);
 if (i == 37) {
  break OuterLoop;
 }}}}
```

Calculator Program

```
import java.awt.*;
import java.awt.event.*;
public class calculator extends java.applet.Applet implements ActionListener {
TextField txtTotal = new TextField("");
Button button[] = new Button[10];
Button divide = new Button("/");
Button mult = new Button("*");
Button plus = new Button ("+");
Button minus = new Button("-");
Button isequalto = new Button("=");
Button clear = new Button("CA");
double num ,numtemp ;
 int counter;
 String strnum = "",strnumtemp = "" ;
 String op = "";
  public void operation() {
    counter ++;
        if (counter == 1) {
         numtemp = num;
         strnum = "";
        num = 0;
         }else{
          if (op == "+") numtemp += num;
          else if (op == "-") numtemp -= num;
         else if (op == "*") numtemp = numtemp * num;
         else if (op == "/") numtemp = numtemp / num;
          strnumtemp = Double.toString(numtemp);
          txtTotal.setText(strnumtemp);
          strnum = "";
          num = 0;
         }
    }
  public void init() {
     setLayout(null);
     plus.setBackground(Color.blue);
     plus.setForeground(Color.white);
     minus.setBackground(Color.blue);
     minus.setForeground(Color.white);
     divide.setBackground(Color.blue);
     divide.setForeground(Color.white);
     isequalto.setBackground(Color.blue);
     isequalto.setForeground(Color.white);
     mult.setBackground(Color.blue);
     mult.setForeground(Color.white);
     clear.setBackground(Color.blue);
     clear.setForeground(Color.red);
  for(int i = 0;i <= 9; i ++) {
        button[i] = new Button(String.valueOf(i));
```

```
        button[i].setBackground(Color.orange);
         button[i].setForeground(Color.blue);
      }
      button[1].setBounds(0,53,67,53);
      button[2].setBounds(67,53,67,53);
      button[3].setBounds(134,53,67,53);
      button[4].setBounds(0,106,67,53);
      button[5].setBounds(67,106,67,53);
      button[6].setBounds(134,106,67,53);
      button[7].setBounds(0,159,67,53);
      button[8].setBounds(67,159,67,53);
      button[9].setBounds(134,159,67,53);
   for (int i = 1;i <= 9; i ++) {
         add(button[i]);
      }
      txtTotal.setBounds(0,0,200,53);
     add(txtTotal);
      plus.setBounds(0,212,67,53);
      add(plus);
      button[0].setBounds(67,212,67,53);
      add(button[0]);
      minus.setBounds(134,212,67,53);
      add(minus);
      divide.setBounds(134,264,67,53);
      add(divide);
      isequalto.setBounds(67,264,67,53);
      add(isequalto);
      mult.setBounds(0,264,67,53);
      add(mult);
      add(clear);
     }
     public void start() {
      for(int i = 0;i <= 9; i ++) {
         button[i].addActionListener(this);
      }
      plus.addActionListener(this);
      minus.addActionListener(this);
     divide.addActionListener(this);
      mult.addActionListener(this);
      isequalto.addActionListener(this);
      clear.addActionListener(this);
    }
     public void stop() {
      for(int i = 0;i <= 9; i ++) {
     button[i].addActionListener(null);
      }
      plus.addActionListener(null);
      minus.addActionListener(null);
      divide.addActionListener(null);
      mult.addActionListener(null);
```

```
 isequalto.addActionListener(null);
 clear.addActionListener(null);
}
public void actionPerformed(ActionEvent e) {
    for(int i = 0;i <= 9; i++) {
        if (e.getSource() == button[i]) {
           play(getCodeBase(),i + ".au");
          strnum += Integer.toString(i);
           txtTotal.setText(strnum);
           num = Double.valueOf(strnum).doubleValue();
        }
    }
    if (e.getSource() == plus) {
     operation();
     op = "+";
    }
   if (e.getSource() == minus) {
     operation();
    op = "-";
  }
    if (e.getSource() == divide) {
     operation();
     op = "/";
    }
    if (e.getSource() == mult) {
     operation();
     op = "*";
    }
   if (e.getSource() == isequalto) {
     if (op == "+") numtemp += num;
     else if (op == "-") numtemp -= num;
    else if (op == "*") numtemp = numtemp * num;
     else if (op == "/") numtemp = numtemp / num;
     strnumtemp = Double.toString(numtemp);
     txtTotal.setText(strnumtemp);
    strnumtemp = "";
     numtemp = 0;
     strnum = "";
     num = 0;
    counter = 0;
    }
    if (e.getSource() == clear) {
     txtTotal.setText("0");
     strnumtemp = "";
     numtemp = 0;
     strnum = "";
    num = 0;
     counter = 0;
    }
} }
```

cal.html

```
<html>
<applet code=”calculator.class” width=400 height=400>
</applet>
</html>
```

Q. What is the purpose of Text Field. List and explain it’s constructors and important methods.

Ans : Text Field: This is also the text container component of Java AWT package. This component contains single line and limited text information. This is declared as follows :

TextField txtfield = new **TextField**(20);

we can fix the number of columns in the text field by specifying the number in the constructor. In the above code we have fixed the number of columns to 20.

TextField Example

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class TextFieldDemo extends Applet implements ActionListener
  {
  TextField name,pass;
  public void init()
   {
   Label namep=new Label("Name: ",Label.RIGHT);
   Label passp=new Label("Password:",Label.RIGHT);
   name=new TextField(12);
   pass=new TextField(8);
   add(namep);
   add(name);
   add(passp);
   add(pass);
   name.addActionListener(this);
   pass.addActionListener(this);
   }
 public void actionPerformed(ActionEvent ae)
   {
   repaint();
   }
 public void paint(Graphics g)
   {
   g.drawString("Name: "+name.getText(),6,60);
   g.drawString("Selected text in name :"+name.getSelectedText(),6,80);
   g.drawString("Password:" +pass.getText(),6,100);
}}
```

Event Handling

Events :

In the delegation event model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Event listeners

A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

Note : The methods that receive and process events are defined in a set of interfaces found in java.awt.event.

Event Listener Interface

Interface	**Description**
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Define five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
TextListener	Defines one method to recognize when a text value changes.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

The ActionListener interface

This interface defines the actionPerformed() method that is invoked when an action

Event occours.

Syntax : void actionPerformed(ActionEvent ae)

The ItemListener interface

This listener defines the itemStateChanged() method that is invoked when the state

Of an item changes.

Syntax : void itemStateChanged(ItemEvent ie)

The keyListener interface

The interface defines three methods :

void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)

The MouseListener interface

This interface defines five methods.

void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)

The MouseMotionListener interface

This interface defines two methods.

void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)

The TextListener interface

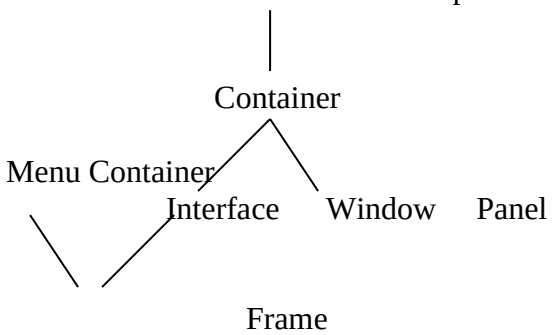
Syntax : void textChanged(TextEvent te)

AWT

AWT stands for **Abstract Window ToolKit.** The Abstract Windowing Toolkit (AWT) is Java's platform-independent windowing, graphics, and user-interface widget toolkit. The AWT is part of the Java Foundation Classes (JFC) - the standard API for providing a graphical user interface (GUI) for a Java program.

Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specify 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 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 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. Other container objects can be stored inside of a Container. A Container is responsible for laying out (that is positioning) any components that it contains.

Panel

The Panel class is a concrete subclass of Container. It doesn't add new methods; it simply implements Container.

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, we won't create Window objects directly. Instead, we will use a subclass of Window called Frame.

Frame

Frame encapsulates what is commonly thought of as a "window". It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners. When a Frame window is created by a program rather than an applet, a normal window is created.

Working with Frame Windows

After the applet, the type of window we will most often create is derived from Frame. We will use it to create child windows within applets, and top-level or child windows for applications.

There are two types of Frame constructors :

a) Frame() -> It creates a standard window that does not contain a title.

b) Frame(String title) -> It creates a window with the title specified by title. Here, we can't specify the dimensions of the window. Instead, we must set the size of the window after it has been created.

Setting the window's Dimensions

a) setSize() -> It is used to set the dimensions of the window.

Void setSize(int newWidth, int newHeight)

Void setSize(Dimension newSize)

The new size of the window is spefied by newWidth and new Height, or by the width and height fields of the Dimension object passed in newSize. The dimensions are specified in terms of pixels.

Hiding and Showing a window

After a frame window has been created, it will not be visible until you call setVisible().

void setvisible(boolean visibleflag)

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

Setting a window's Title

We can change the title in a frame window using **setTitle()** method.

void setTitle(String newTitle)

Here, newTitle is the new title for the window.

Closing a Frame window

When using a frame window, our program must remove that window from the screen when it is closed, by calling **setVisible(false)**. To intercept a window-close event, we must implement the **windowClosing()** method of the **WindowListener** interface.

AWT Controls

Controls are components that allow a user to interact with your application and the AWT supports the following types of controls:

Labels, Push Buttons, Check Boxes, Choice Lists, Lists, Scrollbars, Text Components. These controls are subclasses of Component.

1) **Labels :** This is the simplest component of Java Abstract Window Toolkit. This component is generally used to show the text or string in your application and label never perform any type of action. Syntax for defining the label only and with justification :

   Label label_name = new **Label** ("This is the label text.");

   Above code simply represents the text for the label.

   Label label_name = new **Label** ("This is the label text.", **Label.CENTER**);
   Justification of label can be left, right or centered. Above declaration used the center justification of the label using the Label.CENTER.

2) **Buttons :** This is the component of Java Abstract Window Toolkit and is used to trigger actions and other events required for your application. The syntax of defining the button is as follows :

   Button button_name = new **Button** ("This is the label of the button."); we can change the Button's label or get the label's text by using the **Button.**setLabel(String) and **Button.**getLabel() method. Buttons are added to the it's container using the add (button_name) method.

3) **Check Boxes :** This component of Java AWT allows you to create check boxes in your applications. The syntax of the definition of Checkbox is as follows :

   CheckBox checkbox_name = new **Checkbox** ("Optional check box 1", false);

Above code constructs the unchecked Checkbox by passing the boolean valued argument *false* with the Checkbox label through the Checkbox() constructor. Defined Checkbox is added to it's container using add (checkbox_name) method. we can change and get the checkbox's label using the setLabel (String) and getLabel() method. we can also set and get the state of the checkbox using the setState(boolean) and getState() method provided by the **Checkbox** class.

4) **Radio Button :** This is the special case of the Checkbox component of Java AWT package. This is used as a group of checkboxes which group name is same. Only one Checkbox from a Checkbox Group can be selected at a time. Syntax for creating radio buttons is as follows :

CheckboxGroup chkgp = new **CheckboxGroup**();

add (new **Checkbox** ("One", chkgp, false);

add (new **Checkbox** ("Two", chkgp, false);

add (new **Checkbox** ("Three",chkgp, false);

In the above code we are making three check boxes with the label "One", "Two" and "Three". If you mention more than one true valued for checkboxes then your program takes the last true and show the last check box as checked.

5) **Text Area:** This is the text container component of Java AWT package. The Text Area contains plain text. TextArea can be declared as follows:

   TextArea txtArea_name = new **TextArea**();
   we can make the Text Area editable or not using the setEditable (boolean) method. If you pass the boolean valued argument *false* then the text area will be non-editable otherwise it will be editable. The text area is by default in editable mode. Text are set in the text area using the setText(string) method of the **TextArea** class.

6) **Text Field:** This is also the text container component of Java AWT package. This component contains single line and limited text information. This is declared as follows :

   TextField txtfield = new **TextField**(20);
   You can fix the number of columns in the text field by specifying the number in the constructor. In the above code we have fixed the number of columns to 20.

The Menu components

There are two types pop up and pull down. Pull down menus are accessed via a menu bar, which may contain multiple menus. Menu bars may appear only in frames.

To create a frame with a menu bar containing a pull down menu:

- Create a menu bar and attach it to frame.
- Create and populate the menu.
- Attach the menu to the menu bar.

To create a menu bar, instantiate MenuBar class. To attach it to frame, pass it into the frame's setMenuBar().

To create a menu, instantiate Menu class. The most common constructor takes a string that is the menu's label. There are four kinds of elements that can be mixed and matched to populate a menu:

- MenuItems
  - A menu item is an ordinary textual component available on a menu. The basic constructor is
    MenuItem(String text) where text is the label of the menu item.
    A menu item is very much like a button that happens to live in a menu. Like buttons, menu items generates action events.
- Checkbox MenuItems
  - A checkbox menu item looks like a menu item with a checkbox to the left of its label. When a checkbox menu item is selected, the checkbox changes its state. The basic constructor is *CheckboxMenuItem (String label)*;. You can read and set an item's state by calling getState() and *setState*(). These generate ItemEvents.
- Separators
  - A separator is just a horizontal mark used for visually dividing a menu into sections. To add a separator to a menu, call the menu's addSeparator() method().
- Menus
  - When you add a menu to another menu, the first menu's label appears in the second menu, with a pull-right icon. Pulling the mouse to the right causes the sub-menu to appear.

After a menu is fully populated, you attach it to a menu bar by calling menu bar's add() method. If you want the menu to appear in the Help menu position to the right of all other menus, call instead setHelpMenu().

Menu Example

```
//example on Menu Components
//contains MenuBar,Menus and MenuIems
//menuexample.java

import java.awt.*;
import java.awt.event.*;

public class menuexample extends Frame implements ActionListener
{
      MenuBar mb;
        Menu  m1,m2,m3;
        MenuItem mi1,mi2,mi3,mi4,mi5,mi6,mi7,mi8;
```

```
public menuexample()
{
        mb = new MenuBar();
        m1 = new Menu("File");
        m2 = new Menu("Edit");
        m3 = new Menu("Quit");

        mi1 = new MenuItem("New...");
        mi2 = new MenuItem("Open");
        mi3 = new MenuItem("Save");

        m1.add(mi1);
        m1.add(mi2);
        m1.add(mi3);

        mi4 = new MenuItem("cut");
        mi5 = new MenuItem("copy");
        mi6 = new MenuItem("paste");

        m2.add(mi4);
        m2.add(mi5);
        m2.add(mi6);

        mi7 = new MenuItem("Help");
        mi8 = new MenuItem("Exit");

        m3.add(mi7);
        m3.add(mi8);

        mb.add(m1);
        mb.add(m2);
        mb.add(m3);

        setMenuBar(mb);

        setTitle("menu example....");
        setVisible(true);
        setSize(300,300);
        addWindowListener(new WindowAdapter()
        {
                public void windowClosing(WindowEvent we)
                {
                        System.exit(0);
                }
        });
 mi8.addActionListener(this);
} //constructor
public void actionPerformed(ActionEvent ae)
{
   System.exit(0);
```

```
        }
        public static void main(String[] args)
        {
                menuexample me = new menuexample();
        }
}
```

What is a Layout Manager?

A layout manager is an object that controls the size and position (layout) of components inside a Container object. For example, a window is a container that contains components such as buttons and labels. The layout manager in effect for the window determines how the components are sized and positioned inside the window.

List of Layout Managers

The java.awt package provides the following predefined layout managers that implement the java.awt.LayoutManager interface. Every Abstract Window Toolkit (AWT) and Swing container has a predefined layout manager as its default. It is easy to use the container.setLayout method to change the layout manager, and you can define your own layout manager by implementing the `java.awt.LayoutManager` interface. This article describes the predefined AWT layout managers in this list.

- java.awt.BorderLayout
- java.awt.FlowLayout
- java.awt.CardLayout
- java.awt.GridLayout
- java.awt.GridBagLayout

BorderLayout

The border layout is the default layout manager for all Window objects. It consists of five fixed areas: North, South, East, West, and Center. You do not have to put a component in every area of the border layout. The demonstration puts the label "Border Layout" in the North area, the OK button in the South area, and the List of fruit trees in the Center area. The East (right) and West (left) areas are empty.

FlowLayout

The flow layout is the default layout manager for all Panel objects and applets. It places the panel components in rows according to the width of the panel and the number and size of the components. 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.

GridLayout

The Grid layout arranges components into a grid of rows and columns. we specify the number of rows and columns, the number of rows only and let the layout manager determine the numer of columns, or the number of columns only and let the layout manager determine the number of rows.

GRAPHICS CONTEXTS AND GRAPHICS OBJECTS

A Graphics object manages a graphics context by controlling how objects are drawn. Graphics objects contain methods for drawing, font manipulation, color manipulations, and the other related operations. It has been developed using the Graphics object g (the argument to the applet's paint method) to manage the applet's graphics context. In other words, wou can say that Java's Graphics is capable of:

• Drawing 2D shapes
• Controlling colors
• Controlling fonts
• Providing Java 2D API
• Using More sophisticated graphics capabilities
• Drawing custom 2D shapes
• Filling shapes with colors and patterns.

Graphics Context and Graphics Class

• Enables drawing on screen
• Graphics object manages graphics context
• Controls how objects are drawn
• Class Graphics is abstract
• Cannot be instantiated
• Contributes to Java's portability
• Class Component method paint takes Graphics object.

The `Graphics` class is the abstract base class for all graphics contexts. It allows an application to draw onto components that are realized on various devices, as well as on to off-screen images.

Graphics Objects
In Java, all drawing takes place via a `Graphics` object. This is an instance of the class `java.awt.Graphics`.
Initially the Graphics object we use will be passed as an argument to an applet's paint() method. The drawing can be done by using Applet Panels, Frames, Buttons, Canvases, etc.
Each `Graphics` object has its own coordinate system, and methods for drawing strings, lines, rectangles, circles, polygons, etc. Drawing in Java starts with particular `Graphics` object. we get access to the Graphics object through the *paint(Graphics g)* method of your applet.
Each draw method call will look like
g.drawString("Hello World", 0, 50);
where g is the particular Graphics object with which you're drawing.

Color Control

It is known that Color enhances the appearance of a program and helps in conveying meanings. To provide color to our objects use class Color, which defines methods and

constants for the manipulation of colors. Colors are created from **red**, **green** and **blue** components **RGB** values.

All three RGB components can be integers in the range 0 to 255, or floating point values in the range 0.0 to 1.0
The first part defines the amount of *red*, the second defines the amount of *green* and the third defines the amount of *blue*. So, if we want to give a dark red color to our graphics we will have to give the first parameter value 255 and two parameter zero.

Some of the most common colors are available by name and their RGB values.

Table 1: Colors and their RGB values Color Constant	**Color**	**RGB Values**
Public final static Color ORANGE	Orange	255, 200, 0
Public final static Color PINK	Pink	255, 175, 175
Public final static Color CYAN	Cyan	0, 255, 255
Public final static Color MAGENTA	Magenta	255, 0, 255
Public final static Color YELLOW	Yellow	255, 255, 0
Public final static Color BLACK	Black	0, 0, 0
Public final static Color WHITE	White	255, 255, 255
Public final static Color GRAY	Gray	128, 128, 128
Public final static Color LIGHT_GRAY	light gray	192, 192, 192
Public final static Color DARK_GRAY	dark gray	64, 64, 64
Public final static Color RED	Red	255, 0, 0
Public final static Color GREEN	Green	0, 255, 0
Public final static Color BLUE	Blue	0, 0, 255

As we do with any variable, we should preferably give our colors descriptive names. For instance

```
Color medGray = new Color(127, 127, 127);
Color cream = new Color(255, 231, 187);
Color lightGreen = new Color(0, 55, 0);
```

we should note that Color is not a property of a particular rectangle, string or other object we may draw. Color is a part of the `Graphics` object that does the drawing. we change the color of our `Graphics` object and everything we draw from that point forward will be in the new color, at least until we change it again.

When an applet starts running , its color is set to *black by default*. we can change this to red by calling *`g.setColor(Color.red)`*. we can change it back to black by calling *`g.setColor(Color.black)`*.

Fonts
Unlike HTML Java allows you to choose your fonts. Java implementations are guaranteed to have a *serif font* like *Times* that can be accessed with the name *"Serif"*, a monospaced font like *courier* that can be accessed with the name *"Mono",* and a *sans serif* font like *Helvetica* that can be accessed with the name *"SansSerif"*.

GridBagLayout

The Grid bag layout (like grid layout) arranges components into a grid of rows and columns, but lets you specify a number of settings to fine-tune how the components are sized and positioned within the cells. Unlike the grid layout, the rows and columns are not constrained to be a uniform size. For example, a component can be set to span multiple rows or columns, or you can change its position on the grid.
GridBagConstraints gbc = new GridBagConstraints();

gridx and gridy

The `gridx` and `gridy` fields specify the x and y coordinates of the cell at the upper left of the `Component's` display area. The upper-left-most cell has coordinates (0, 0). The mnemonic constant `GridBagConstraints.RELATIVE` specifies that the Component is placed immediately to the right of (gridx), or immediately below `(gridy)` the previous Component added to this container.

Gridwidth and Gridheight

The `gridwidth` and `gridheight` fields specify the number of cells in a row `(gridwidth)` or column `(gridheight)` in the Component's display area. The mnemonic constant `GridBagConstraints.REMAINDER` specifies that the `Component` should use all remaining cells in its row `(for gridwidth)` or column (for `gridheight`). The mnemonic constant `GridBagConstraints`.RELATIVE specifies that the Component should fill all but the last cell in its row `(gridwidth)` or column `(gridheight)`.

Fill

The `GridBagConstraints` `fill` field determines whether and how a component is resized if the component's display area is larger than the component itself. The mnemonic constants you use to set this variable are
`GridBagConstraints.NONE` : Don't resize the component
`GridBagConstraints.HORIZONTAL`: Make the component wide enough to fill the display area, but don't change its height.
`GridBagConstraints.VERTICAL`: Make the component tall enough to fill its display area, but don't change its width.
`GridBagConstraints.BOTH`: Resize the component enough to completely fill its display area both vertically and horizontally.

Ipadx and Ipady

Each component has a minimum width and a minimum height, smaller than which it will not be. If the component's minimum size is smaller than the component's display area, then only part of the component will be shown.

The `ipadx` and `ipady` fields let you increase this minimum size by padding the edges of the component with extra pixels. For instance setting `ipadx` to 2 will guarantee that the component is at least 4 pixels wider than its normal minimum. (`ipadx` adds 2 pixels to each side.)

Insets

The `insets` field is an instance of the `java.awt.Insets` class. It specifies the padding between the component and the edges of its display area.

weightx and weighty

The `weightx` and `weighty` fields determine how the cells are distributed in the container when the total size of the cells is less than the size of the container. With weights of zero (the default), the cells all have the minimum size they need, and everything clumps together in the center. All the extra space is pushed to the edges of the container.

Swing

The Swing toolkit includes a rich set of components for building GUIs and adding interactivity to Java applications. Swing includes all the components you would expect from a modern toolkit: table controls, list controls, tree controls, buttons, and labels.

Swing is part of the Java Foundation Classes (JFC). The JFC also include other features important to a GUI program, such as the ability to add rich graphics functionality and the ability to create a program that can work in different languages and by users with different input devices.

The following list shows some of the features that Swing and the Java Foundation Classes provide.

Swing GUI Components

The Swing toolkit includes a rich array of components: from basic components, such as buttons and check boxes, to rich and complex components, such as tables and text. Even deceptively simple components, such as text fields, offer sophisticated functionality, such as formatted text input or password field behavior. There are file browsers and dialogs to suit most needs, and if not, customization is possible. If none of Swing's provided components are exactly what you need, you can leverage the basic Swing component functionality to create your own.

Pluggable Look-and-Feel Support

Any program that uses Swing components has a choice of look and feel. The JFC classes shipped by Sun and Apple provide a look and feel that matches that of the platform. The Synth package allows you to create your own look and feel. The GTK+ look and feel makes hundreds of existing look and feels available to Swing programs.

Difference between AWT and Swing

AWT - Heavy weight component. Every graphical units it will invoke native methods.
SWING - Light weight component. It doesn't invoke native methods.

Swing is based on MVC architetcure(Model view Controller) and thats why its look and feel is independent of hardware and OS...whereas AWT look and feel depends upon the platform...

Swing are the extension of Awt.
Awt have some disadvanage that are removed by he swing .for example
awt programs are not machine independent.
it causes some problem on different machine
whereas his does not occur in swing.

Swing provides some advanced compoent like JTree JTabbedPane JList and many more

Swing GUI components are packaged into Package javax.swing. In the Java class hierarchy there is a class
Class Component which contains a method paint for drawing Component onscreen
Class Container which is a collection of related components and contains method add for adding components and Class JComponent which has
Pluggable look and feel for customizing look and feel
Shortcut keys (*mnemonics*)
Common event-handling capabilities
The Hierarchy is as follows:

Object----- Component--Container---JComponent

In Swings, we have classes prefixed with the letter 'J' like

JLabel -> Displays single line of read only text

JTextField -> Displays or accepts input in a single line

JTextArea -> Displays or accepts input in multiple lines

JCheckBox ->Gives choices for multiple options

JButton -> Accepts command and does the action

JList - > Gives multiple choices and display for selection

JRadioButton - > Gives choices for multiple option, but can select one at a time.

Registration.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class Registration1 extends JFrame implements ActionListener,ItemListener
{
public Registration1()
        {
        display();
        }
        JPanel         panel;
        JLabel         label1,label2,label3,label4,label5,label6,
                      label7,label8,label9,label10,label11,
                      label12,
                       label13,label14,label15,label16;
        JTextField     text1,text2,text3,text4,text5,text6,text7,
                       text8,text9,text10,text11,text12,text13,
                       text14;
        JButton              bt1,bt2,bt3,bt4,bt5;
        JComboBox  cb1,cb2;
        public void display()
                {
                setTitle("Registration From...");
                setSize(1024,600);
                setVisible(true);
                panel=new JPanel();
                getContentPane().add(panel);
                panel.setLayout(null);
//-------------------------PATIENT ID-----------------------------------
label1=new JLabel();
label1.setText("Patient ID");
panel.add(label1).setBounds(100,50,160,35);
label1.setFont(new Font("courier new",Font.BOLD,18));
label1.setForeground(new Color(50,48,20));
//-------------------------TEXT1----------------------------------------
text1=new JTextField(20);
text1.setBackground(new Color(10,255,160));
panel.add(text1).setBounds(300,50,150,30);
//-------------------------REG...CATAGARY-------------------------------------
label2=new JLabel();
label2.setText("Reg...Catagary");
panel.add(label2).setBounds(100,85,160,35);
label2.setFont(new Font("courier new",Font.BOLD,18));
label2.setForeground(new Color(50,48,20));
//-------------------------TEXT2----------------------------------------
text2=new JTextField(20);
text2.setBackground(new Color(10,255,160));
panel.add(text2).setBounds(300,85,150,30);
//-------------------------PATIENT TYPE-----------------------------------
label3=new JLabel();
```

```
label3.setText("Patient Type");
panel.add(label3).setBounds(100,120,160,35);
label3.setFont(new Font("courier new",Font.BOLD,18));
label3.setForeground(new Color(50,48,20));
//-------------------------TEXT3-----------------------------------
text3=new JTextField(20);
text3.setBackground(new Color(10,255,160));
panel.add(text3).setBounds(300,120,150,30);
//-------------------------DEPARTMENT---------------------------------
label4=new JLabel();
label4.setText("Department");
panel.add(label4).setBounds(100,155,160,35);
label4.setFont(new Font("courier new",Font.BOLD,18));
label4.setForeground(new Color(50,48,20));
//-------------------------TEXT4-----------------------------------
text4=new JTextField(20);
text4.setBackground(new Color(10,255,160));
panel.add(text4).setBounds(300,155,150,30);
//-------------------------CONSULTANT ID-----------------------------------
label5=new JLabel();
label5.setText("Consultant ID");
panel.add(label5).setBounds(100,190,160,35);
label5.setFont(new Font("courier new",Font.BOLD,18));
label5.setForeground(new Color(50,48,20));
//-------------------------TEXT5-----------------------------------
text5=new JTextField(20);
text5.setBackground(new Color(10,255,160));
panel.add(text5).setBounds(300,190,150,30);
//-------------------------FIRST NAME-----------------------------------
label6=new JLabel();
label6.setText("First Name");
panel.add(label6).setBounds(100,225,160,35);
label6.setFont(new Font("courier new",Font.BOLD,18));
label6.setForeground(new Color(50,48,20));
//-------------------------TEXT6-----------------------------------
text6=new JTextField(20);
text6.setBackground(new Color(10,255,160));
panel.add(text6).setBounds(300,225,150,30);
//-------------------------MIDDLE NAME-----------------------------------
label7=new JLabel();
label7.setText("Middle Name");
panel.add(label7).setBounds(100,260,160,35);
label7.setFont(new Font("courier new",Font.BOLD,18));
label7.setForeground(new Color(50,48,20));
//-------------------------TEXT7-----------------------------------
text7=new JTextField(20);
text7.setBackground(new Color(10,255,160));
panel.add(text7).setBounds(300,260,150,30);
//-------------------------LAST NAME-----------------------------------
label8=new JLabel();
```

```
label8.setText("Last Name");
panel.add(label8).setBounds(100,295,160,35);
label8.setFont(new Font("courier new",Font.BOLD,18));
label8.setForeground(new Color(50,48,20));
//-------------------------TEXT8-----------------------------------
text8=new JTextField(20);
text8.setBackground(new Color(10,255,160));
panel.add(text8).setBounds(300,295,150,30);
//-------------------------DATE------------------------------------
label9=new JLabel();
label9.setText("Date");
panel.add(label9).setBounds(550,50,160,35);
label9.setFont(new Font("courier new",Font.BOLD,18));
label9.setForeground(new Color(50,48,20));
//-------------------------TEXT9-----------------------------------
text9=new JTextField(20);
text9.setBackground(new Color(10,255,160));
panel.add(text9).setBounds(740,50,150,30);
//-------------------------D.O.B-----------------------------------
label10=new JLabel();
label10.setText("D.O.B");
panel.add(label10).setBounds(550,85,160,35);
label10.setFont(new Font("courier new",Font.BOLD,18));
label10.setForeground(new Color(50,48,20));
//-------------------------TEXT10----------------------------------
text10=new JTextField(20);
text10.setBackground(new Color(10,255,160));
panel.add(text10).setBounds(740,85,150,30);
//-------------------------GENDER----------------------------------
label11=new JLabel();
label11.setText("Gender");
panel.add(label11).setBounds(550,120,160,35);
label11.setFont(new Font("courier new",Font.BOLD,18));
label11.setForeground(new Color(50,48,20));
//-------------------------COMBOBOX1-------------------------------
cb1=new JComboBox();
cb1.addItem("Male");
cb1.addItem("Female");
cb1.addItemListener(this);
cb1.setFont(new Font("courier new",Font.BOLD,16));
cb1.setBackground(new Color(10,255,160));
panel.add(cb1).setBounds(740,120,150,30);
//-----------------------------MARITAL

STATUS----------------------------------
label12=new JLabel();
label12.setText("Marital Status");
panel.add(label12).setBounds(550,155,160,35);
label12.setFont(new Font("courier new",Font.BOLD,18));
label12.setForeground(new Color(50,48,20));
```

```
//---------------------------COMBOBOX2-----------------------------------
cb2=new JComboBox();
cb2.addItem("Married");
cb2.addItem("Unmarried");
cb2.addItemListener(this);
cb2.setFont(new Font("courier new",Font.BOLD,16));
cb2.setBackground(new Color(10,255,160));
panel.add(cb2).setBounds(740,155,150,30);

//-------------------------OCCUPATION-----------------------------------
label13=new JLabel();
label13.setText("Occupation");
panel.add(label13).setBounds(550,190,140,35);
label13.setFont(new Font("courier new",Font.BOLD,18));
label13.setForeground(new Color(50,48,20));
//---------------------TEXT11-------------------------------------
text11=new JTextField(20);
text11.setBackground(new Color(10,255,160));
panel.add(text11).setBounds(740,190,150,30);
//-------------------------ADDRESS-------------------------------------
label14=new JLabel();
label14.setText("Address");
panel.add(label14).setBounds(550,225,140,35);
label14.setFont(new Font("courier new",Font.BOLD,18));
label14.setForeground(new Color(50,48,20));
//-------------------------TEXT12-------------------------------------
text12=new JTextField(20);
text12.setBackground(new Color(10,255,160));
panel.add(text12).setBounds(740,225,150,30);
//-------------------------MOBILE-------------------------------------
label15=new JLabel();
label15.setText("Mobile");
panel.add(label15).setBounds(550,260,140,35);
label15.setFont(new Font("courier new",Font.BOLD,18));
label15.setForeground(new Color(50,48,20));
//-------------------------TEXT13-------------------------------------
text13=new JTextField(20);
text13.setBackground(new Color(10,255,160));
panel.add(text13).setBounds(740,260,150,30);
//-------------------------REMARKS-------------------------------------
label16=new JLabel();
label16.setText("Remarks");
panel.add(label16).setBounds(550,295,140,35);
label16.setFont(new Font("courier new",Font.BOLD,18));
label16.setForeground(new Color(50,48,20));
//-------------------------TEXT14-------------------------------------
text14=new JTextField(20);
text14.setBackground(new Color(10,255,160));
panel.add(text14).setBounds(740,295,150,30);
//-------------------------ADD BUTTON-------------------------------------
```

```
bt1=new JButton("ADD");
panel.add(bt1).setBounds(100,450,110,40);
bt1.setFont(new Font("courier new",Font.BOLD,18));
bt1.setForeground(new Color(255,78,90));
bt1.setBackground(new Color(50,48,20));
//---------------------------CLEAR BUTTON-------------------------------------
bt2=new JButton("CLEAR");
panel.add(bt2).setBounds(250,450,110,40);
bt2.setFont(new Font("courier new",Font.BOLD,18));
bt2.setForeground(new Color(255,78,90));
bt2.setBackground(new Color(50,48,20));
//-------------------------SEARCH BUTTON-------------------------------------
bt3=new JButton("SEARCH");
panel.add(bt3).setBounds(400,450,110,40);
bt3.setFont(new Font("courier new",Font.BOLD,18));
bt3.setForeground(new Color(255,78,90));
bt3.setBackground(new Color(50,48,20));
//-------------------------EDIT BUTTON---------------------------------------
bt4=new JButton("EDIT");
panel.add(bt4).setBounds(550,450,110,40);
bt4.setFont(new Font("courier new",Font.BOLD,18));
bt4.setForeground(new Color(255,78,90));
bt4.setBackground(new Color(50,48,20));
//-------------------------DELETE BUTTON-------------------------------------
bt5=new JButton("DELETE");
panel.add(bt5).setBounds(700,450,110,40);
bt5.setFont(new Font("courier new",Font.BOLD,18));
bt5.setForeground(new Color(255,78,90));
bt5.setBackground(new Color(50,48,20));
setDefaultCloseOperation(EXIT_ON_CLOSE);
}
          public void actionPerformed(ActionEvent ae)
          {
          }
       public void itemStateChanged(ItemEvent ie)
       {
       }
          public static void main(String arg[])
          {
          Registration1 from=new Registration1();
          from.display();
          }
}
```