JAVA BASICS

Java is a general purpose, object oriented programming language developed by Sun MicroSystems in 1991. Originally called OAK by James gosling , one of the inventors of the language. But was renamed as java in 1995.

1. Features of Java:

I. **Simple**: Java was designed to be easy for professional programmers to learn, if he already knows C &C++. Java includes syntaxes from C and object oriented concepts from C++. The confusing concepts in both C & C++ are leftover here. So java is easy to learn and it is simple language.

II. **Secure**: Security becomes an important issue for a language that is used for programming on internet. Threat of viruses and abuse of resources are everywhere. Java systems not only verify all memory access but also ensure that no viruses are communicated with an applet. The absence of pointers in java ensures that programs can't gain access to memory locations without proper authorization.

III. **Portable**: The most significant contribution of java over other languages is its portability. Java programs can be easily moved from on computer to another, anywhere and anytime. Changes and upgrades in operating systems, processors and system resources will not force any changes in java programs. This is the reason why java has become popular language for programming on internet which interconnects different kinds of systems worldwide.

IV. **Object-oriented**: java is a true object oriented language. Almost everything in java is an object. All program code and data reside with in objects & classes. Java comes with an extensive set of classes, arranged in packages, that we can use in our programs by inheritance. The object Model in java is simple and easy to extend.

V. **Robust**: Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and runtime checking for data types. It is designed as a garbage collected language relieving the programmers virtually all memory management problems. Java also incorporates the concept of exception handling which captures serious errors and eliminates any risk of crash the system.

VI. **Multithreaded**: It means handling multiple tasks simultaneously. Java supports multithreaded programs. This means that we need not wait for the application to finish one task before another. For eg, we can listen to an audio clip while scrolling a page and at the same time download an applet from a distant computer. This feature greatly improves the interactive performance of graphical applications.

VII. **Architectural Neutral:** One of the problems facing by programmers was program written today will not be run tomorrow even in the same machine or if the OS or if the processor upgrades. So java designers made it architectural neutral by implementing JVM (Java Virtual Machine)through java runtime environment. The main role of java designers to make it architecture neutral write once, run anywhere, anytime forever.

VIII. **Compiled & Interpreted:** Usually a computer language is either compiled or interpreted. Java combines both these approaches thus making java a two stage system. First java compiler translates source code into what is known as byte code. Byte codes are not machine instructions and therefore, in second stage java interpreter generates machine code that can be directly executed by the machine that is running the java program. So we can say that java is both compiled and interpreted language.

IX. **High Performance:** Java performance is impressive for an interpreted language, mainly due to the use of intermediate byte code. Java architecture is also designed to

reduce overheads during runtime. Further, the incorporation of multithreading enhances the over all execution speed of java programs.

X. **Dynamic:** java programs carry with them substantial amount of runtime information that is used to access the objects at runtime. This makes java Dynamic.

XI. **Distributed:** java is designed for the distributed environment of the internet because it handle TCP/IP protocols. Java supports two computers to support remotely through a package called Remote Method Invocation (RMI)

2. What are the data types in java?

Data types:
Every variable in java has a data type. Data types specify the size and type of values that can be stored. Java language is rich in its data types.
Integer types:

Java supports 4 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 +ve or –ve

Type	Size	Range
byte	1 byte	-128 to 127
short	2 bytes	-2^{15} to $2^{15}-1$
int	4 bytes	-2^{31} to $2^{31}-1$
long	8 bytes	-2^{63} to $2^{63}-1$

byte: The smallest integer type is byte. This is a signed 8-bit type that has a range from -128 to 127. Variables of type 'byte' are especially useful when you're working with a stream of data from a n/w or file. byte variables are declared by use of 'byte' keyword.

eg: byte b, c;

short: short is signed 16 b it type. It has range from -2^{15} to $2^{15}-1$. The short variables are declared by use of short keyword.

eg: short s;

int: The most commonly used integer type is int. It is a signed 32 bit type that a range from -2^{31} to $2^{31}-1$. In addition to other uses, variables of type int are commonly employed to control loop and to index arrays. 'int' type is most versatile & efficient type. int variables are declared by use of 'int' keyword.

eg: int a;

long: long is a signed 64 bit type and is useful for those occasions where an int type is not large enough to hold the desired values. The range of long is quite large. It has range from -2^{63} to $2^{63}-1$. long variables are declared by use of 'long' keyword

eg: long a;

Floating point types:
Floating point numbers are used when evaluating expressions that require fractional

precision. There are two kinds of floating point types, float and double. Which represent single and double precision numbers respectively.

Type	Size	Range
float	4 bytes	1.4e-045 to 3.4e+038
short	8 bytes	4.9e-324 to 1.8e+308

float: the type float specifies a single precision value that uses 32 bits of storage. Single precision is faster on same processors and takes half as much space as double precision. float variables are declared by use of 'float' keyword.

eg: float a=1.234;

double: double precision, as denoted by the double keyword, uses 64 bits to store a value. All transcendental math functions such as sin(), cos() and sqrt() return double values. double variables are declared by use of 'double' keyword.

eg: double d;

char: In java, the datatype used to store characters is char. Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. Thus in java char is a 16bit type. The range of char is 0 to 65536. There are no negative chars. char variables are declared by use of char keyword.

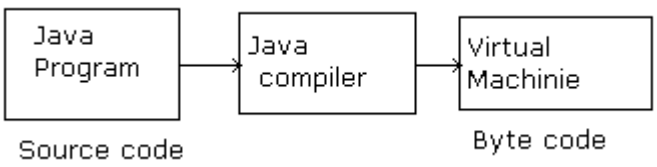
eg: char c;

boolean: Java has a simple type called boolean, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators such as a<b. Boolean is also the type required by conditional expressions that govern the control statements such as 'if' and 'for'. boolean variables are declared by use of 'boolean' keyword.

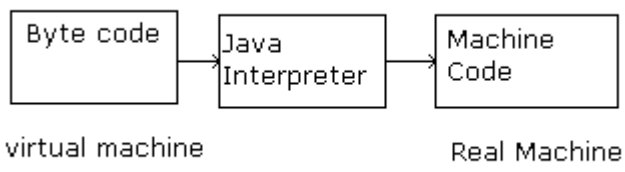
eg: boolean b; b=false;

3. Java Virtual Machine:

Generally all language compilers translate source code into machine code for a specific computer. In java, architectural neutral is achieved because compiler produces an intermediate code called Byte code, for a machine is called JVM



A virtual machine code is not machine specific. The code generated by the java interpreter by acting as an intermediary between the virtual machine and real machine.



4. **Implementing a java program:**

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

- Creating the program
- Compiling the program
- Running the program

Remember that, before we begin creating the program, JDK must be properly installed on our system.

Creating the program:

We can create a program using any text editor

```
class Sample
{
        public static void main(String arg[])
        {
                System.out.println("Hi friend");
        }
}
```

We have to save this program with a name like prog.java

A java Application have any no of classes, but only one main class. Main class is nothing but class which contains main() method

Compiling the program:

To compile the program we must run the java compiler javac, with the name of the source file on command line as shown below

```
C:\>    javac prog.java
```

If everything is OK, java compiler(javac) creates a file called Sample.class (<classname.class>) containing the bytecodes of the program prog.java

Running the program:

We need to use the java interpreter to run stand alone applications. At the command prompt, type…

```
C:\> java Sample
```

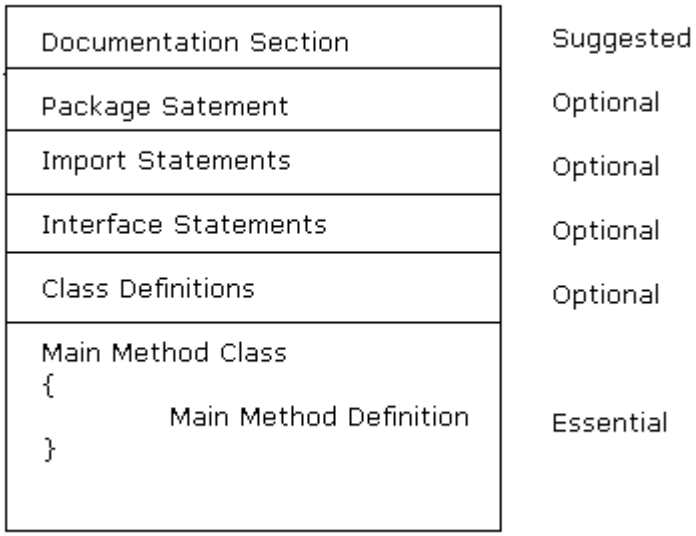
Now, the interpreter looks for the main method in the program and begins execution from there. When executed our program displays the following

```
o/p:    Hi friend
```

5. **Java PROGRAM STRUCTURE:**

A java program may contain many classes of which only one class defines a main method. Classes contain datamembers and methods that operate on the data members of the class.

Java program may contain one or more sections as shown below.



Documentation section:
This section comprises of a set of comment lines giving the name of program, the author & other details, which the programmer would like to refer at later stage. It is suggested one.
Package statement:
The first statement allowed in java is a 'package' statement.
This statement declares a package name and informs the compiler that the classes defined here belong to this package.

Eg: package student;

It is optional section.
Import statement:
This is similar to the #include statement in C
Eg: import java.io.*;
It is optional section.
Interface statement:
An interface is like a class but includes a group of method declarations. It is also optional section and is used only when we wish to implement the multiple inheritance feature in jprogram.
Class Definitions:
A java program may contain multiple class definitions. Classes are the primary & essential elements of a java program.
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.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 operating system.

6. COMMAND LINE ARGUMENTS:

Sometimes we will need to supply data into a java program when it runs. This type of data will be supplied in the form of command line arguments. The command line arguments will immediately follow the program name on the command line. These command line arguments will be collected into an array of string which is supplied as a parameter to main methods.

Later these command line arguments must be processed and then accessed. The following program takes command line argument and prints them.

Eg:
```
class Display
    {
        public static void main(String args[])
        {
            for(i=0;i<args.length();i++)
                System.out.println(args[i]);
        }
    }
```
c:\> javac file.java

c:\> java Display hi this is ur friend

o/p-> hi
this
is
ur
friend

7. VARIABLES:

A variable is an identifier that denotes a storage location used to store a data value. All variables have a scope, w hich defines their visibility and a lifetime.

Declaring variable:

All variables must be declared before they can be used. The basic form of a variable declaration is , type identifier[=value][, identifier [=value]…];

Eg: int value;

float avg;

in the above example, 'value' is an integer type variable , where as 'avg' is float type variable.

To declare more than one variable of the specified type, use a comma-separated list.

```
int a,b,c;        // declares three integer variables    a, b, and c
int d=3,e,f=5;  // declares 3 integer variables initializing d and f
```

Unlike in C, variable can be declared any where in the program, where ever they are needed.

Dynamic Initialization:

Although the preceding examples have used only constants as initializers, java allows

variables to be initialized dynamically, using any expression valid at the time the variable is declared.
For example,

```
Class   DynInit
{
        Public static void main(String arg[])
        {
                double a=3.0, b=4.0;
                double c=Math.sqrt(a*a+b*b);                    //   c is dynamically
initialized
                System.out.println("Hypotenuse is  "+c);
        }
}
```

Here, three local variables a, b, c are declared. The two a and b are initialized by constants. However c is initialized dynamically.

8. Classification of variables:

Java variables are actually classified into three kinds:

- Instance variables
- Class variables
- Local 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.

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 variable.

- Instance and class variables are declared inside a class.

Local variables are declared and used inside methods and program blocks(defined between 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.

9. Scope and lifetime of variables:

Scope of variables: Scope is defined as an area of the program where the variable is accessible. As a general rule, variables declared insid a scope are not visible to code that is defined outside that scope.

Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope.This means that objects declared in the outer scope will be visible to code within the inner scope. However the reverse is not true. Objects declared with in the inner scope will not be visible outside it.

// program to demostarte block scope

```
class scope
{
    public static void main(String ar[])
    {
        Int  x; // known to all code with in main
```

```
        x=10;
            if(x==10)
            {
                Int  y =20;  // known only to this block
                System.out.println("x and y: "+x+" "+y);
                x=y*2;
            }
        y = 100; // Error! y not known here
        System.out.println("x is "+x);              // x is still known here
        }
}
```

As the comments indicate, the variable x is declared at the start of **main()**'s scope and is accessible to all subsequent code within main(). y is only visible to the code in if block .

Lifetime of variables:

Varaibles are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also a variable declared within a block will lose its values when the block is left. Thus, the lifetime of a variable is confined to its scope.

```
class Lifetime
{
    public static void main(String ar[])
        {
                int x;
                for(x=0; x<3; x++)
                {
                        int y=-1;                   // y is initialized each time block
    is entered
                        System.out.println("y is : "+y);                    // this
    always prints -1
                        Y=100;
                        System.out.println("y is now : "+y);
                }
        }
}
```

The output of above program is :

```
y is : -1
y is now:  100
y is : -1
y is now:  100
y is : -1
y is now:  100
```

10. Arrays

An array is a collection of homogeneous data items that share a common name. Arrays of

anytype can be created and may have one or more dimensions. A specific element in an array is accessed by its index.

One-Dimensional Arrays:
general form of a one-dimensional array declaration is

type var-name[];

eg: int marks[];

and we must use **new** operator to allocate memory. So the general form of new as it applies to one-dimensional arrays appears as follows.

type var-name[]=new type[size];

eg: int marks[] = new int[20];

```
//Demonstrate a one-dimensional array
class Array
{
	public static void main(String ar[])
	{
	int month_days[]=new int[12];
	month_days[0] = 31;
	month_days[1] = 28;
	month_days[2] = 31;
	month_days[3] = 30;
	month_days[4] = 31;
	month_days[5] = 30;
	month_days[6] = 31;
	month_days[7] = 31;
	month_days[8] = 30;
	month_days[9] = 31;
	month_days[10] = 30;
	month_days[11]= 31;
	System.out.println("October has  "+month_days[9]);
	}
}
```

We can also write above program as below…

```
class Array
{
	public static void main(String ar[])
	{
	int month_days[]={31,28,31,30,31,30,31,31,30,31,30,31};
	System.out.println("October has  "+month_days[9]+" days");
	}

	}
}
```

Two (Multi)-Dimensional Arrays:

In java, multidimensional arrays are actually arrays of arrays.

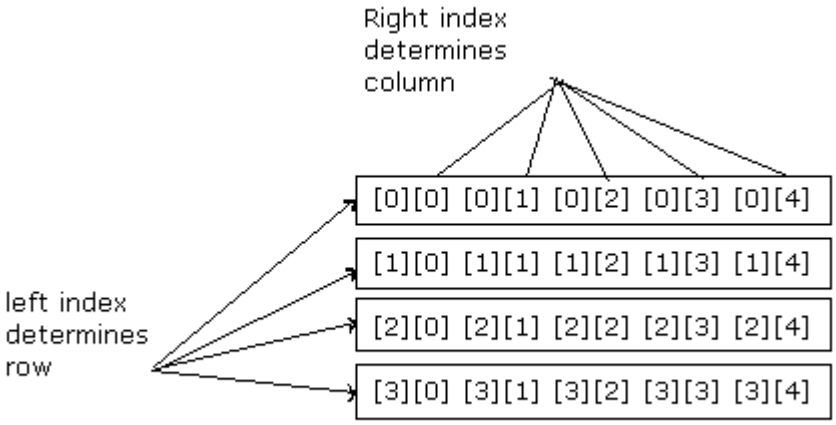
example:

```
int  twoD[][] = new int[4][5];
```

eg program:

```
class Matrix
{
	public static void main(String ar[])
	{
		int mat[][]=new int[3][4];
		for(int i=0;i<3;i++)
		{
			for(int j=0;j<4;j++)
			{
				mat[i][j]=i;
				System.out.print(mat[i][j]+" ");
			}
			System.out.println();
		}
	}
}
```

the above program inserts values into a matrix 'mat' of order 3x4 and prints




```
// Manually allocate differing size second dimensions
class Traingle
{
	public static void main(String ar[])
	{
		int t[][] = new int[4][];
		t[0] = new int[1];
		t[1] = new int[2];
		t[2] = new int[3];
		t[3] = new int[4];
	for(int i=0;i<4;i++)
		{
			for(int j=0;j<i+1;j++)
			{
				t[i][j]=i;
				System.out.print(t[i][j]+" ");
			}
			System.out.println();
		}
```

```
    }
}
```

11. Operators

Java supports a rich set of operators. Operators are used in programs to manipulate data and variables. Java operators can be classified into a number of related categories as below:

1. **A**rithmetic operators
2. **I**ncrement & Decrement operators
3. **R**elational operators
4. **B**itwise operators
5. **L**ogical operators
6. **A**ssignment Operator
7. **C**onditional Operator

1)Arithmetic operators:

These are used in mathematical expressions in the same way that they are used in algebra. The following are the list of arithmetic operators:

Operator	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo division (Remainder)
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

Arithmetic operators are used as shown below:

```
a-b     a+b
a*b     a/b
a%b     -a*b
```

- here a and b are operands

2)Increment & Decrement operators:

The ++ and the – are java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one.
for example,

```
x=x+1;  can be written like x++;  using increment operator
similarly,
x=x-1;  can be written like x--;  using decrement operator
```

3)Relational operators:

The relational operators determine the relationship that one operand has to the other. The outcome of these operators is a **boolean** value. The relational operational operators are most frequently used in the expressions that control the **if** statement and the various **loop** statements. The following are the list of Relational operators:

Operator	**Meaning**
= =	Equal to

! =	Not equal to
>	Greater than
<	Less than
> =	Greater than or equal to
< =	Less than or equal to

As stated, the result produced by a relational operator is a Boolean value. For example,

```
int a=4;
int b=1;
boolean c=a>b;
```

- in this case, the result of a<b(which is true) is stored in c

```
if(a>b)
{
}
```

- in this case, a>b results true, so if statement will be executed

Bitwise operators:

Java defines several bitwise operators which can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands. The following are the list of Bitwise operators:

Operator	**Meaning**
&	bitwise AND
!	bitwise OR
^	bitwise XOR
~	bitwise unary NOT
< <	shift left
> >	shift right
> > >	shift right with zero fill

Logical operators:

Logical operators are also called as Boolean Logical operators. They operate on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	**Meaning**
&&	logical AND
||	logical OR
^	logical XOR
!	logical NOT

The logical operators && and || are used when we want to form compound conditions by combining two or more relations.

for example,

```
if( a>b && x ==10) { }
```

A	**B**	**A&&B**	**A||B**	**A^B**	**!A**
true	true	true	true	false	false
true	false	false	true	true	false
false	true	false	true	true	true
false	false	false	false	false	true

Assignment Operator (=):

The assignment operator is the single equal sign, = . It has general form:

var =expression;

Here, the type of var must be compatible with the type of expression.

the assignment operator allows to create a chain of assignments.

For example,

```
int  x, y, z;
x = y = z = 100;  // set x, y and z to 100
```

Conditional Operator(**?:**)

The character pair ?: is a ternary operator available in Java. This operator is used to construct conditional expressions of the form

exp1 ? exp2 : exp3

exp1 is evaluated firt. It it is nonzero (true), then the exp2 is evaluated and becomes the value of the conditional expression. if exp1 is false, exp3 is evaluated and its value becomes the value of the conditional expression.

For example,

```
a=10 , b=20;
x = ( a > b )? a : b ;
```

x will be assigned the value of b. This can be achieved using the **if…else** statement as follows:

```
if(a > b)
        x = a;
else
        x = b;
```

12. Expressions

When operands and operators are combined, an expression is formed. The execution of an expression will produce a value.

Arithmetic expression:

If we use arithmetic operators in an expression, that is treated as an Arithmetic expression.

Eg: a+b-c;

Increment or Decrement Expression.

If we use Increment or Decrement operators in an expression, that is treated as Increment or Decrement expression. Eg: a++; b- -;

Relational expression:

If we use Relational operators in an expression, that is treated as a Relational expression: Eg: a>b;

Logical expression:

If we use Boolean logical operators in an expression, that is treated as a logical expression:

Eg: a&&b;

Conditional expression:

If we use arithmetic operators in an expression, that is treated as an Conditional expression.

Eg: a>b? a: b;

… like this, we have some other expressions.

13. Control Statements

A Control statement is a statement that controls the flow of execution of the program. Java's program control statements are divided into 3 categories

- Selection Statements
- Iteration Statements
- Jump Statements

1)Selection Statements:

Selection statement controls the flow of the program depending on the result of the conditional expression or the state of a variable. There are two selection statements : **if** and **switch**

a. if statement:
'if' is a selection statement that is used to choose two choices in a program.
Syntax:

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

'condition' is any expression that returns a Boolean value (i.e., true/false). Statement is a single or multiple statements.

If the condition is true then the control goes to the statement1 and it is executed. Otherwise, the statement2 is executed.

b. Nested if statement:
It means an if statement under another if statement.
Syntax:

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

        }
```

c. if-else-if ladder:
if-else-if statement is a sequence of if-else statements.
Syntax:

```
if(condition)
        statement1;
else if(condition)
        statement2;
else if(condition)
        statement3;
.
.
else    statement2;
```

In if-else-if ladder if a condition is not met then the control flows from top to bottom until the last if statement.

d. switch statement:
It provides more than one choice to choose. It's a better alternative to if-else-if ladder.
Syntax:

```
switch(expression)
{
case value1:  statement1;
```

```
                    break;
case value2:  statement2;
                    break;
case value3:  statement3;
                    break;
                    .
                    .
                    .
case valueN:  statementN;
                    break;
default :         statement;
}
```

here, the expression may be of type byte, short, int or char only. the case values must be of type expression and each value should be unique. when control comes to the switch statement then the value of the expression is compared with the case values, if a match is found then the statement(s) corresponding to that case is executed. If none of the case is matched then the default statement is executed.

2)Iteration Statements:

These statements allows the part of the program to repeat one or more times until some condition becomes true. There are 3 iteration statements: **while**, **do-while** and **for**

a. while statement:
'while' is an iteration statement, that repeats a statement or block of statements until some condition is true.
Syntax:

```
while(condition)
{
// body of the loop
}
```

where the condition may be any boolean expression. The body of the loop will be executed as long as the condition is true. Once the condition becomes false, the control passes to the statement immediately after the while loop.

b. do-while statement:
'do-while' statement is very much similar to while statement with little difference. In while statement, if the condition is initially false then the body of loop will not be executed at all. Where as in do-while statement, body of the loop will be executed at least once since the condition of do-while is at the bottom of the loop.
Syntax:

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

Each iteration of do-while executes the body of loop first and evaluates the condition later. If condition is true, the loop repeats. Otherwise, the loop terminates.

c. for statement:
'for' statement also repeats the execution of statements while its condition is true.
Syntax:

```
for(initialization; condition; iteration)
{
// body of the loop
}
```

where initialization portion of the loop sets the value of the variable that acts as a counter. This portion is executed first when the loop starts. And it is executed only once. when the control goes to condition, condition is evaluated. If condition is true, the body of loop is executes. Otherwise the loop terminates.Next iteration portion is executed. This helps to increment or decrement the control variable.

The loop then iterates evaluating condition, then executing the body and then executing the iteration portion each time it iterates.

2) Jump Statements:

Jump statements allows the control to jump to a particular position. There are 3 types of jump statements.

a. break statement:
in java, there are 3 uses of break statement.
i. It helps to terminate a statement sequence in **switch** statement.
ii. It can be used to exit or **terminate** the loop
iii. It can be used as another form of **goto**

Eg:

```
class A
{
        public static void main(String arg[])
        {
                for(int i=0; i<100; i++)
                {
                        if(i==10)
                        break; // terminates loop if i is 10
                        System.out.println("i :"+i);
                }
        }
}
```

b. continue statement:
The continue statement starts the next iteration of the immediately enclosing iteration statements(while, do-while or for). When the control goes to continue statement, then it skips the remaining statements and starts the next iteration of enclosing structure.

Eg:

```
class A
{
        public static void main(String arg[])
        {
                for(int i=0; i<100; i++)
```

```
                {
                        System.out.println("i :"+i);
                        if(i%2==0)
                           continue;
                        System.out.println("  ");
                }
        }
}
```

c. return statement:
This is used to return from a method. When the control reaches to return statement, it causes the program control to transfer back to the caller of the method.

Eg:

```
class A
{
        public static void main(String arg[])
        {
                boolean t=true;
                if(t)
                return;
                System.out.println("Hi"); // This won't execute
        }
}
```

In this example, return statement returns to the java run time system, since the caller of main() is runtime system.

14. Type Conversion & Costing

We can assign a value of a variable of one type to a variable of another type. This process is known as casting. If two types are compatible with each other, then the type conversion is done implicitly by Java. This is known as automatic (implicit) type conversion.
Eg:

```
int a=10;

double b=a;
```

As shown in above example, it is always possible to assign the value of int variable 'a' to double variable 'b'. This is done implicitly by the system as both variable's datatypes are compatible

conversion of larger datatype to smaller datatype is known as Narrowing

conversion of smaller datatype to larger datatype is known as Widening.

Automatic Type conversion:
The system performs automatic type conversion when one type of data is assigned to another type of variable only if following rules are satisfied.

i. The types are compatible
ii. The destination type is larger than the source type.

A widening conversion took place when these 2 rules are satisfied. Example, widening conversion takes place between int and byte as int is larger than byte to hold its all valid values.

15. Casting incompatible types:

What if we want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is called narrowing conversion. To create a conversion between two incompatible types, we must use a cast. A cast is simply an explicit type conversion. It has this general form:

(target-type) value

Here, target-type specifies the desired type to convert the specified value to.

The following program demonstrates some type conversions that require casts:

```
class conversion
{
	public static void main(String arg[])
	{
		byte b;
		int i= 257;
		double d=323.142;

		System.out.println(”Conversion of int to byte”);
		b=(byte) i;
		System.out.println(“i and b ”+i+” ”+b);

		System.out.println(”Conversion of double to int”);
		i=(int) d;
		System.out.println(“i and b ”+i+” ”+b);

		System.out.println(”Conversion of double to byte”);
		b=(byte) d;
		System.out.println(“i and b ”+i+” ”+b);

	}
}
```

this program generates the following result

Conversion of int to byte

i and b 257 1

Conversion of double to int

d and i 323.142 323

Conversion of double to byte

d and b 323.142 67

In above eg, When the value 257 is cast into a byte variable, the result is the remainder of the division of 257 by 256(range of byte), which is 1 in this case. When the d is converted to an int, its fractional component is lost. When d is converted to a byte, its fractional component is lost, and the value is reduces modulo 256, which in this case is 67.

16. Classes & objects:

Concept of Classes

A class is defined as an encapsulation of data and methods that operate on the data. When we are creating a class we are actually creating a new datatype. This new datatype is also called ADT.

--diagram-

When we define a class we can easily create a no of instances of that class

Creating class in java:
In java, a class can be defined through the use of class keyword. The general definition of a class is given below:

```
class classname
{
        type instance_variable1;
        type instance_variable1;
        .
        .
        .
        returntype method name(parameter list)
        {
                // body of method
        }
        .
        .
        .
}
```

Here,

- class is a keyword used to define class
- class name is the identifier that specifies the name of the class
- type specifies the datatype of the variable
- instance_variable1, . . . are the variable defined in the class
- method name is the method defined in the class that can operate on the variables in the class

Example:

```
class sample
{
        int x, y;
        setXY()
        {
         x=10; y=20;
        }
}
```

The variables defined in the class are called member variables/data members
The functions defined in the class are called member functions/member methods.
Both data members and member methods together called members of class.

17. Concept of Objects

Objects are instances of a class. Objects are created and th is process is called "Instantiation". In java objects are created through the use of new operator. The new operator

creates an object and allocates memory to that object. Since objects are created at runtime, they are called runtime entities.

In java object creation is a two step process.

Step1: create a reference variable of the class

Eg: Sample s;

When this declaration is made a reference variable is created in memory and initialized with null.

Step2: create the object using the new operator and assign the address of the object to the reference variable created in step1.

Eg: Sample s=new Sample();

The step2 creates the object physically and stores the address of the object in the reference variable.

18. Accessing members of an object:

Once an object of a class is created we can access the members of the class through the use of object name and '.' (dot) operator.

Syntax: objectname. member;

Example:

```
class sample
{
        int x, y;
        setXY()
        {
         x=10; y=20;
        }
        printXY()
        {
                System.out.print("="+x);
                System.out.print("="+y);

        }
}
class Demo
{
        public static void main(String ar[])
        {
                Sample s;
                s=new String();
                s.setXY();
                s.printXY();
        }
}
```

19. Constructors

A constructor is a special kind of method that has the same name as that of class in which it is defined. It is used to automaticallyl initialize an object when the object is created. Constructor gets executed automatically at the time of creating the object. A constructor

doesn't have any return type.

Example:

```
// program to demonstrate constructor

class Student
{
        int number;
        String name;
        Student()       // default constructor
        {
                number=569;
                name="satyam";
        }
        Student(int no, String s)       // parameterized  constructor

        {
                number=no;
                name=s;
        }
        void showStudent()
        {
                System.out.println("number ="+number);
                System.out.println("name ="+name);
        }
}

class Demo
{
        public static void main(String ar[])
        {
        Student s1=new Student();
        Student s2=new Student(72,"raju");
        s1.showStudent();
        s1.showStudent();
        }
}

output:number= 569
         name= satyam
         number= 72
         name= raju

// and explain about above program
```

20. 'this' keyword

'this' keyword refers to the current object. sometimes a member method of a class needs to refer to the object in which it was invoked. In order to serve this purpose java introduces "this keyword".

By using 'this' keyword we can remove name space conflict.

Name Space Conflict:

When the parameters of a member method have the same name as that of instance variables of the class, local variables take the priority over the instance variable. This leads to conflict called 'name space conflict'. It can be resolved through use of "this"
eg:

```
class Student
{
 int number;
String name;
        void setStudent(int number,  String name)
        {
                this.number=number;
                this.name=name;
        }
        void showStudent()
        {
                System.out.println("number="+number);
                System.out.println("name="+name);
        }
}
class Display
{
        public static void main(String ar[])
        {
                Student s=new Student(); // object creation for class Student
                s.setStudent(1,"Rama");
                s.showStudent();
        }
}
```

Understanding **static** keyword:
In java , static keyword can be used for the following purposes.

- To create static data members
- To define static methods
- To define static blocks

21. Static variables:

When a datamember of a class, is declared static only one copy of it is created in memory and is used by all the objects of the class. Hence static variables are essentially global variables.

When a modification is performed on a static variable the change will be reflected in all objects of the class for which the static variable is a member.

22. Static Methods:

- A static method can call other static methods only
- A static method can access static variables only
- Static methods cant have access to this/ super keyword.

Static Blocks:

Just like static variables and static methods we can also create a static block . this static block is used to initialize the static variables.
Example program:

```
class UseStatic
{
        static int a=3;
        static int b;
        static void meth(int x)
        {
                System.out.println("x=" + x);
                System.out.println("a=" + a);
                System.out.println("b=" + b);

        }
        static
        {
                System.out.println("static block initialized");
        }

        public static void main(String args[])
        {
                meth(69);
        }
}

        output:
                static block initialized
                x = 69
                a = 3
                b = 12

        // and explain about program
```

23. Access Control:

Java's access specifiers are public, private and protected. Java also defines a default access level. protected applies only when inheritance is involved.
When a member of a class is specified as public, then that member can be accessed by anyother code.
When a member of class is specified as private, then that member can only be accessed by other members of its class, but not anyother class members.

24. Garbage Collection

Objects are dynamically allocated using the 'new' operator and their memory must be released after reallocation. In C++, they are manually released by the use of delete operator. Java deallocates memory automatically and it is called "Garbage Collection". When no references to an object exist the object is assumed to be no longer needed and the memory occupied by the object can be reclaimed and t his is how "garbage collection" is done.

25. Finalize() Method :

In some situations an object will lead to perform some action when it is destroyed. Suppose for example an object is holding some non-java resource such as file-handling. We must have to free the resources after the object is destroyed.

To handle these situations java provides finalization using finalize() method

Syntax:
```
protected void finalize()
{
        // finalization code here
}
```
protected prevents to access finalize method by code defined outside the class.

void is used because it doesn't return anything.

26. Overloading methods

In java, it is possible for a class, to contain two or more methods with the same name as long as the method signatures are different. That is the method should have different number of parameters or different type of parameters when this is the case the methods are said to be overloaded and the mechanism is called method overloading.

Example:

```
class Student
{
        int  no ,marks;
        String name;
        void setStudent()
        {
                no=1;
                marks=89;
                name="rama";
        }

        void setStudent(int  no, int marks, String name)
        {
                this.no=no;
                this.marks=marks;
                this.name=name;
        }
        void showStudent()
        {
                System.out.println("number="+no);
                System.out.println("marks="+marks);
                System.out.println("name"+name);
        }
}
class MethodOverload
{
        public static void main(String ar[])
        {
                Student s=new Student();
                s.setStudent();
                s.showStudent();

                s.setStudent(69,81,"ramu");
                s.showStudent();
        }
```

```
}
```
output:

```
number=1
marks=89
name=rama

number=69
marks=81
name=ramu
```

27. Overloading Constructors

Just like member methods constructors can also be overloaded. the following example demonstrates this,

```
class Student
{
    int no;
    String name;
    Student()
    {
        no=70;
        name="ram";
    }
    Student(int n, String s)
    {
        no=n;
        name=s;
    }
    void show()
    {
        System.out.println("Number="+no);
        System.out.println("Name="+name);
    }
}
class Display
{
    public static void main(String ar[])
    {
        Student s1= new Student();
        Student s2= new Student(69,"satya");
        s1.show();
        s2.show();
    }
}
```

output:

```
number=70
name=ram
number=69
name=satya
```

28. Parameter passing

There are two ways that a computer language can pass an argument to a subroutine (method).

1. call by value
2. call by reference

In the first way, the change made to the parameter of the subroutine have no effect on the argument.

In the second way, the changes made to the parameter will effect the argument used to call the subroutine.

In java when you pass a primitive type it is pass by value. when you pass an object(reference) to a method, it is done through pass by reference

Example:

```
        // call by value
class Test
{
        void meth(int i , int j)
        {
                i+=2;
                 j*=2;
        }
}
class CallByValue
{
        public static void main(String arg[])
        {
                Test ob= new Test();
                int a=10 , b=20;
                System.out.println("a and b before call: "+a+" "+b);
                        ob.meth(a,b);
                System.out.println("a and b after  call: "+a+" "+b);

        }
}
```

output:

```
        a and b before call: 10 20
        a and b after  call: 10 20
```

29. Recursion

Java supports recursion. It is the process of defining something interms of itself. It is the attribute that allows a method to call itself. The method that calls itself is said to be recursive method.

The classic example of recursion is the computation of the factorial of a number.

```
class Factorial
{
        int fact(int n)
        {
```

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

output above program:

Factorial of 5 is 120

30. Nested & inner classes:

It is possible to define a class with in another class. Such classes are known as nested classes. The scope of nested class is bounded by the scope of its enclosing class. Thus, if class B is defined with in class A, then B is known to A, but not outside of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

There are two types of nested classes: static and non-static. A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it can not refer to members of its enclosing class directly.

The most important type of nested class is the **inner** class. An inner class is a non-static nested class. It has access to all of the variable and methods of its outer class and may refer to them directly.

Example:

```
        class  Outer
        {
          int outer_x=10;
            void outerMethod()
            {
                System.out.println("In Outerclass Method");
                Inner in=new Inner();
                In.innerMethod();
            }
            class Inner
            {
                void innerMethod()
```

```
                {
                        System.out.println("In Innerclass Method");
                        System.out.println("outer class variable:"+outer_x);
                }
        }
}

class Display
{
        public static void main(String ar[])
        {
                Outer out=new Outer();
                Out.outerMethod();
        }
}
```

output:

```
In Outerclass Method
In Innerclass Method
outer class variable: 10
```

31. String Handling

A String is a sequence of characters. In java , Strings are class objects and implemented using two classes, namely, **String** and **StringBuffer.** A java string is an instantiated object of the String class.A java String is not a character array and is not NULL terminated. Strings may be declared and created as follows:

String stringname= new **String**("string");

String name=new String("kanth"); is same as
String name="kanth";

like arrays, it is possible to get the length of string using the length method of the String class.

```
int len = name.length();
```

Java string can be concatenated using the + operator.

```
eg:     String firstname = "sri";
        String lastname = "kanth";
        String name = firstname+lastname;
                        ( or )
        String name = "sri"+"kanth";
```

32. String Arrays:

we can also create an use arrays that contain strings. The statement,

String names[]=new String[3];

will create an **names** array of size 3 to hold three string constants.

33. String Methods: (Methods of **String** class)

The String class defines a number of methods that allow us to accomplish a variety of string manipulation tasks.

Method	**Task**
s2=s1.toLowerCase()	converts the String s1 to all lowecase
s2=s1.toUpperCase()	converts the String s1 to all Uppercase
s2=s1.replace('x','y');	Replace all appearances of x with y
s2=s1.trim();	Remove white spaces at the beginning and end of String s1
s1.equals(s2);	Returns 'true' if s1 is equal to s2
s1.equalsIgnoreCase(s2)	Returns 'true' if s1=s2, ignoring the case of characters
s1.length()	Gives the length of s1
s1.CharAt(n)	Gives nth character of s1
s1.compareTo(s2)	Returns –ve if s1<s2, positive if s1>s2, and zero if s1 is equal s2
s1.concat(s2)	Concatenates s1 and s2
s1.indexOf('x')	Gives the position of the first occurrence of 'x' in string s1
s1.indexOf('x',n)	Gives the position of 'x' that occurs after nth position in the- string s1

```
//Alphabetical ordering of strings
class StringOrdering
{
	public static void  main(String args[])
	{
		String names[]={"india","usa","australia","africa","japan"};
		int size=names.length;
		String temp;
		for(int i=0;i<size;i++)
		{
			for(int j=i+1;j<size;j++)
			{
				if(names[j].compareTo(names[i])<0)
				{
					temp=names[i];
```

```
                                        names[i]=name[j];
                                        names[j]=temp;
                                    }
                                }
                            }
                            for(int i=0;i<size;i++)
                                System.out.println(names[i]);
                    }
}
```

above program produces the following result

```
africa
australia
india
japan
usa
```

34. StringBuffer Class :

StringBuffer is a peer class of String. While **String** creates string of fixed length, **StringBuffer** creates strings of flexible length that can be modified in terms of both length and content. We can insert characters and substrings in the middle of a string, or append another string to the end.

Below, there are some of methods that are frequently used in string manipulations.

Method	**Task**
s1.setCharAt(n,'x')	Modifies the nth character to x
s1.append(s2)	Appends the string s2 to s1 at the end
s1.insert(n,s2)	Inserts the string s2 at the position n of the string s1
s1.setLength(n) n<s1.length() s1 is	sets the length of the string s1 to n. if truncated .if n>s1.length() zeros are added to s1

INHERITANCE

35. Inheritance:

Inheritance is one of the corner stones of Object oriented programming. Because it allows hierarchical classifications. In terminology of java, a class that is inherited is called a super class. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of super class. It inherits all of the instance variables and methods defined by the super class and adds its own, unique elements.

36. Hierarchical abstractions

A powerful way to manage abstraction is through the use of hierarchical classification. This allows to break the complex systems into manageable pieces.

for example, a car is a single object for outside but the car consists of several subsystems such as steering, breaks, sound system, seat belts etc. Again the sound system consists of radio, DVD player etc.

Here we manage the complexity of a car system through the use of hierarchical classification. Here everything we view in this world is an object. The topmost hierarchy for every object is material object.

Base class:

suppose in the above example, car is the base class. and the sub classes are steering, breaks, engine etc

The behaviour of the child class is always an extension of the properties of the associated with the parent class.

37. Subclass:

Consider the relationship associated with the parent class and subclass

- Instances of a sub class must possess all the data areas associated with the parent class.
- Instances of a sub-class must implement through inheritance at least some or all functionalities defined for the parent class.
- Thus an instance of a child class can mime the behaviour of the parent class and should be indistinguishable from the instance of parent class if substituted in similar situation.

This creates a problem because we use so many forms of inheritance. To solve this, we use principle of substitutability .

38. Substitutability :

Def: "The principle of substitutability" says that if we have two classes 'A' and 'B' such that class B is sub-class of A class, it should be possible to substitute instances of class B for instance of class A in any situation with no observable effect.

39. Subtype: The term subtype often refers to a subclass relationship in which the principle of substitutability is maintained.

statically typed languages (C++) place much more emphasis on principle of substitutability than dynamically typed languages (Small Talk)

The reason for this is the statically typed languages tend to characterize objects by their class and dynamically typed languages tend to characterize by behaviour.

That is the subtype us determined in statically typed languages by their class and dynamically typed languages by their behaviour.

40. Forms of inheritance:

Inheritance is used in variety of ways. The following list represents general abstract categories and is not intended to be exhaustive.

1. Sub-classing for **Specialization**:

The most common use of inheritance and sub-classing is for specialization. In this the derived class is specialized form of the parent class and satisfies the specifications of the parent in all relevant aspects. This is the most ideal form of inheritance, something that a

good design should strive for

eg: A class window provides general windowing operations such as moving, resizing etc. A specialized sub class text edit window inherits the window opens. and in addition provides the windows to display textual material

eg:

```
class Parent
{
        int i=10;
        void show()
        {
                System.out.println("i value="+i);
        }
}
class Child extends Parent
{
        // show() will  be called  by Child Object in main class
}
```

2. Sub-classing for **Specification**:

In the classes maintained a certain common interface i.e, they implement the same methods the parent class can be the combination of implemented operations and the operations are differed to child class. The child merely implements behaviour describe but not implemented in the parent. In such classes the parent class is mostly known as abstract specification class.

Eg:

```
abstract class Parent
{
        int i;
        abstract void show(); // only specifications
}
class Child extends Parent
{
        i=20;
        void  show()
        {
                System.out.println("i ="+i);
        }
```

```
}
```

3. Sub-classing for **Construction**:

A class can often inherit , almost all of its desired functionality from the parent class perhaps changing only the names of the methods used to interface to the class are modifying the arguments in a certain fashion. This may be true if the new class and the parent class failed to share the relationship.

eg:

```
class Parent
{
        int i=10;
        void show()
        {
                System.out.println("i value="+i);
        }
}
class Child extends Parent
{
        i=30;
        void  display() // own implementation of child class
        {
                System.out.println("i value="+i);
        }

}
```

4. Sub-classing for **extension**.

In sub-classing for generalization modifies or extends on the existing functionality of an object. But sub-classing for extension acts totally new abilities. In the sub-classing for generalization it must override atleast one method from the parent and here we must atleast extend one method of the parent.

Eg:

```
class Parent
{
        int i=10;
        void show()
        {
```

```
                System.out.println("i value="+i);
        }
}
class Child extends Parent
{
        // show() will  be called  by Child Object in main class
        void  display() // new method is also implemented
        {
                System.out.println("This is child class");
        }

}
```

5. Sub-classing for **limitation**:

Subclassing for limitation occurs when the behaviour of the sub-class is smaller or more restrictive than the behaviour of the parent class.

Eg:

```
class Parent
{
        int i=10;
        void show()
        {
                System.out.println("i value="+i);
        }
        void  display() // new method is also implemented
        {
                System.out.println("This is Parent class");
        }

}
class Child extends Parent
{
        // show() will  be called  by Child Object in main class
        // display() may not use .i.e, all methods of Parent will not be used
```

}

6. subclassing for **Combination**:

A common situation is that a sub class may represent the combination of features from two or more parent classes.

For example a class teaching assistant can be desired from both the parent classes teacher and assistant

41. Benefits of Inheritance:

various benefits of inheritance are

- S/w Reusability:

  Many programmers spend much of their time in rewriting code they have written many times before. So with inheritance code once written can be reused.

- code sharing

  Code sharing occurs at two levels. At first level many users or projects can use the same class. In the second level sharing occurs when two or more classes developed by a single programmer as part of a project which is being inherited from a single parent class. Here also code is written once and reused. This is possible through inheritance.

- Consistency of inheritance:

  When two or more classes inherit from the same superclass we are assured that the behaviour they inherit will be the same in all cases. Thus we can guarantee that interfaces to similar objects are in fact similar.

- software components:

  Inheritance provides programmers the ability to construct reusable s/w components. The goal behind these is to provide applications that require little or no actual coding. Already such libraries and packages are commercially available.

- Rapid Prototyping:

  When a s/w system is constructed largely out of reusable components development can be concentrated on understanding the new and unusual portion of the system. Thus s/w system can be generated more quickly and easily leading to a style of programming known as ‘Rapid Prototyping’ or ‘exploratory programming’

- Polymorphism and Framework:

  Generally s/w is written from the bottom up , although it may be designed from top-down. This is like building a wall where every brick must be laid on top of another brick i.e., the lower level routines are written and on top of these slightly higher abstraction are produced and at last more abstract

elements are generated. polymorphism permits the programmer to generate high level reusable components.

- Information Hiding:

  A Programmer who reuses a software need only to understand the nature of the component and its interface. There is no need to have detailed information of the component and it is information hiding.

42. Costs of Inheritance:

Although they are benefits with object oriented programming we must also consider the cost of inheritance.

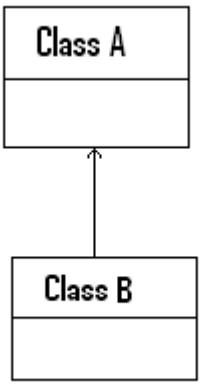
- Execution Speed: The inherited methods which must deal with orbitary sub-classes are often slower than specialized code. Here efficiency is often misplaed. It is far better to develop a working system and monitor it.
- Program Size: The use of any s/w library imposes a size penalty not imposed by systems constructed for a specific project. Although the expense may be substantial and size of program becomes less important.
- Message passing overhead: Message passing is costly by nature a more costly operation than simple procedure invocation
- Program Complexity: Although object oriented programming is often touched as a solution to s/w complexity. the complexity increases when we use inheritance. But it is not considerable.

43. <u>Inheritances in Java:</u>

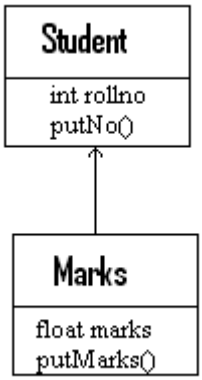
1. Single Inheritance
2. Multi-level Inheritance
3. Hierarchical Inheritance

- Single Inheritance:

  In a class hierarchy when a child has one and only one parent and parent has one only child, that inheritance is said to be single inheritance.



eg:

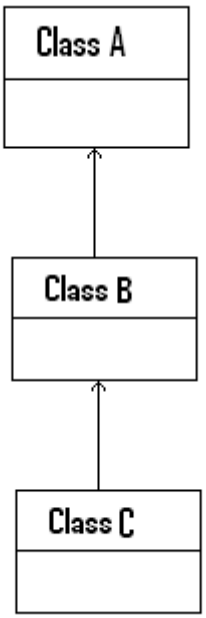



```
class Student
{
        int rollno;
        void getNo(int no)
        {
                rollno=no;
        }
        void putNo()
        {
        System.out.println("rollno= "+rollno);
        }
}
class Marks extends Student
{
        float marks;
        void getMarks(float m)
        {
                marks=m;
        }
        void putMarks()
        {
        System.out.println("marks= "+marks);
        }
}

class Display
```

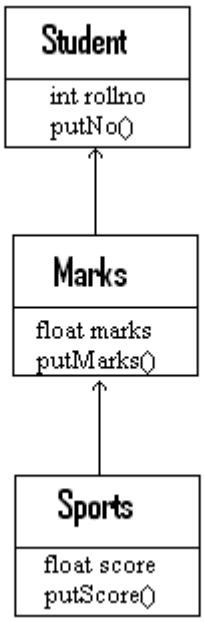
```
{
        public static void main(String ar[])
        {
                Marks ob=new Marks();

                ob.getNo(44);
                ob.putNo();

                ob.getMarks(66);
                ob.putMarks();
        }
}
```

- Multi-level Inheritance

In a class hierarchy, when a class is derived from already derived class then that inheritance is said to be multi-level inheritance.



eg:

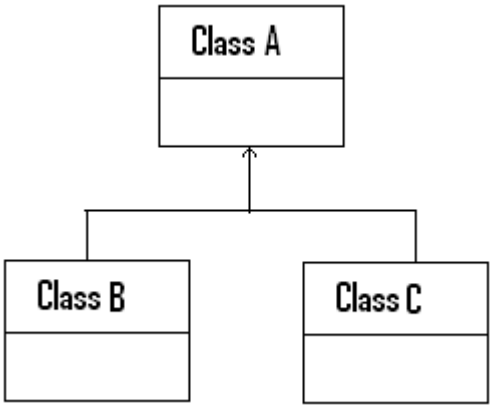



```
class Student
{
        int rollno;

        void getNo(int no)
        {
                rollno=no;
        }

        void putNo()
        {
        System.out.println("rollno= "+rollno);
        }
}
class Marks extends Student
{
        float marks;
        void getMarks(float m)
        {
                marks=m;
        }
        void putMarks()
```

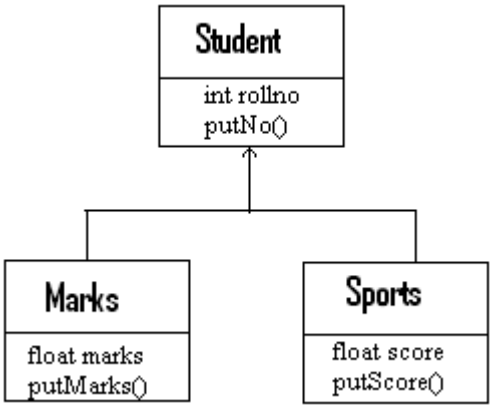
```
        {
        System.out.println("marks= "+marks);
        }
}

class Sports extends Marks
{
        float score;
        void getScore(float scr)
        {
                score=scr;
        }
        void putScore()
        {
        System.out.println("score= "+score);
        }
}

class Display
{
        public static void main(String ar[])
        {
                Sports ob=new Sports();

                ob.getNo(44);
                ob.putNo();

                ob.getMarks(55);
                ob.putMarks();

                ob.getScore(85);
                ob.putScore();
        }
}
```

- Hierarchical inheritance:

In a class hierarchy, when a parent class has two or more than two child classes then, the inheritance is said to be hierarchical inheritance.



e.g:




```
class Student
{
        int rollno;

        void getNo(int no)
        {
                rollno=no;
        }

        void putNo()
        {
        System.out.println("rollno= "+rollno);
        }
}
class Marks extends Student
```

```
{
        float marks;
        void getMarks(float m)
        {
                marks=m;
        }
        void putMarks()
        {
        System.out.println("marks= "+marks);
        }
}

class Sports extends Student
{
        float score;
        void getScore(float scr)
        {
                score=scr;
        }
        void putScore()
        {
        System.out.println("score= "+score);
        }
}

class Display
{
        public static void main(String ar[])
        {
                Marks s1=new Marks();
                Sports s2=new Sports();

                s1.getNo(44);
                s1.putNo();
                s1.getMarks(45);
```

```
                s1.putMarks();

                s2.getNo(44);
                s2.putNo();
                s2.getScore(95);
                s2.putScore();
            }
        }
```

44. Polymorphism:

It is a mechanism of having multiple forms. In other words polymorphism is a mechanism of defining an interface to represent a general class actions.

45. Method Overriding:

In a class hierarchy when a method of a subclass has the same name and type signature has that of a method in its superclass, then the method in the subclass is said to be override the method in the superclass. When this is the case the mechanism called "method overriding" and such methods are called overridden methods.

When overridden method is using base class object the base class version is executed and if it is called by subclass, the subclass version is executed.

eg:

```
class A
{
    void callMe()
    {
        System.out.println("I am version of superclass A");
    }
}
class B extends A
{
    void callMe()
    {
        System.out.println("I am version of subclass B");
    }
}
```

```
class Display
{
        public static void main(String ar[])
        {
                B ob=new B();
                ob.callMe(); // B's callMe() will be executed
                                // i.e, A's callMe() overridden by B's callMe()
        }
}
```

Abstract Keyword:

1. To declare abstract methods
2. To declare abstract classes

1. Abstract Method:

   It is a method in which we have only method declaration but it will not have definition or body. Abstract methods aare declared with abstract keyword and terminated by semicolon (**;**)

   Syntax: abstract return_type MethodName(parameter_list)**;**

   - In a class hierarchy when a superclass containing an abstract method. all the subclasses of that superclass must override the base class method.
   - If the child fails to override the superclass abstract method, the child class must also be abstract.
   - Hence abstract method follows the process of method overriding.

2. Abstract Classes:

   A class which consists atleast one abstract method is called abstract class. The class definition must be preceded by abstract keyword.

   Syntax:

```
abstract class className
{
-----
-----
abstract returntype MethodName(parameter_list)
{
        -------
        -------
```

```
}
        -------
        -------
        // it can include concrete methods also
}
```

eg:

```
abstract class Shape
{
        abstract void callMe();
}

class sub extends Shape
{
        void callMe()
        {
         System.out.println("I am sub's Method")
        }
}

class DemoAbstract
{
        public static void main(String ar[])
        {
                sub s=new sub();
                s.callMe();
        }
}
```

46. final keyword:

final keyword is used for the following purposes

1. To define final datamembers

2. To define final methods
3. To define final classes

1. final datamembers:

   when a data member is declared as final, its value can't be changed through out the program. Hence final variables can be treated as 'java constants' Hence when a datamember is declared as final its value must be assigned immediately.

   eg: final double pie=3.14;

2. final methods:

   when a method is declared as final its value cant be overridden at all

   syntax:
   ```
   final returntype MethodName(ParameterList)
   {
       -------
       -------
   }
   ```

When a base class method is declared as final no subclass of that superclass will be allowed to override the base class final method

eg:

```
final void show()
{
    ------
    ------
}
```

3. final class:

   when a class is declared as a final it cant be inherited at all. Hence we cant have subclasses for final classes.

47. super keyword:

- It is used for referring the immediate superclass of the sub class
- super keyword is used to access super class data members from its subclass.

eg:

```
class A
{
    int x;
}
```

```
class B
{
        int x;
        B(int p, int q)
        {
                super.x=p;      // superclass variable is initialized
                x=q;            // subclass variable is initialized
        }
        void display()
        {
                System.out.println("super class variable="+super.x);
                System.out.println("sub class variable="+ x);

        }

}
class DemoSuper
{
        public static void main(String ar[])
        {
                B ob=new B(10,20);
                ob.display();
        }
}
```

48. super uses:

1)super keyword is used to access superclass methods from its subclass

eg: above example

2) super keyword is used to access superclass constructor from its subclass

eg:

```
class A
{
        int x,y;
        A(int m, int n)
        {
```

```
                x=m;
                y=n;
        }
}
class B
{
        int z;
        B(int p, int q, int r)
        {
                super(p,q);     // superclass variable is initialized
                z=r;            // subclass variable is initialized
        }
        void display()
        {
        System.out.println("super class variables="+super.x+" "+super.y);
        System.out.println("sub class variable="+ z);


        }

}
class DemoSuper
{
        public static void main(String ar[])
        {
                B ob=new B(10,20,30);
                ob.display();
        }
}
```

49. <u>Member access rules</u>

As java is an object-oriented programming language, data is given more importance regarding its access. Data is to be protected against unauthorized access. In order to protect data in a well built way, data is organized into three categories.

a. Public data
b. Private data
c. Protected data

public, private & protected are called access specifiers. The following table illustrates the member access rules

- The members declared as public can be accessed any where, any class and any package.
- The members declared as private can be accessed only by the methods of same class where the private members are declared.
- The members declared as protected cannot be accessed in non-subclasses of different packages in rest of the cases they can be accessed. If you want any member to be accessed in the subclasses of other packages they can be declared as protected. These members are being protected from the access facility of non-subclasses of other packages.
- The members declared by nothing, like of no modifier is being placed before a member of a class then, these members can be access only up to package. This occurs by default. Different packages are not allowed to access these no modifier members.

	Public	**Protected**	**No Modifier**	**Private**
Same class	Yes	Yes	Yes	Yes
Same Package **Same class**	Yes	Yes	Yes	No
Same Package **No-sub Classes**	Yes	Yes	Yes	No
Different Package **Sub Classes**	Yes	Yes	No	No
Different Package **Non-Subclasses**	Yes	No	No	No

PACKAGES & INTERFACES

50. Package:

A package is a collection of classes and interfaces which provides a high level of access protection and names space management.

51. Defining & Creating package:

step1: simply include a package command has the first statement in java source file. Any class you declare with in that file will belong to the specified package.

syntax: package packagename;

Eg: package mypack;

step 2: next define the class that is to be put in the package and declare it as public

step 3: now st ore the classname.java file in the directory having name same as package name.

step 4: file is to be compiled, which creates .class file in the directory

java also supports the package hierarchy , which allows to group related classes into a package and then group related packages into a larger package. We can achieve this by specifying multiple names in a pcakge statement, separated by dot .

i.e., package firstpackage.secondpackage;

Accessing package:

A java system package can be accessed either using a fully qualified classname or using import statement. We generally use import statement.

syntax: import pack1[.pack2][.pack3].classname;

or

import pack1. [.pack2][.pack3].*;

here pack1 is the top level package, pack2 is the package which is inside in pack1 and so on. In this way we can have several packages in a package hierarchy. We should specify explicit class name finally. Multiple import statements are valid. * indicates that the compiler should search this entire package hierarchy when it encounters a class name.

52. Understanding CLASSPATH:

the package hierarchy is controlled by CLASSPATH. Until now we have been storing all of our classes in the same unnamed default package. This works because the default current working directory(.) is usually in the class path environmental variable defined for the java runtime system by default. But there are so many unknown problems when packages are involved.

How does the java runtime system know where to look for packages that we create? The answer has twoparts. First, by default, the java runtime system uses the current working directory as its starting point. Thus if our packages in in the current directory, or a subdirectory of the current directory , it will be found. Second, you can specify a directory path by setting the CLASSPATH environmental variable.

eg: create a class called ‘packtest’ in package ‘test’. create a directory called ‘test’ and put packtest.java inside the directory. Now we compile the program and the resultant .class file will be stored in the current working directory. Then execute the program as ;

java test.packtest

53. Importing packages:

java includes import statement to bring certain classes or entire package into visibility. In a java source file import statement occurs immediately following the package statement and before any class definitions.

syntax: import pkg1[.pkg2].(classname / *) ;

here pkg1 is the name of the top level package. pkg2 is the name of the subordinate package separated by (.)

finally classname/ * indicates whether the java compiler should import the entire package or a part of it.

eg: import java.util.Date;

here java is main package, util is subordinate package Date is the class belongs to util package

Example program(user defined package):

packEg.java

```
package siet;
public class packEg
{
    public void display()
    {
        System.out.println("WELCOME TO SWARNANDHRA INSTITUE");
    }
}
```

packuse.java

```
import siet.packEg;
class Display
{
    public static void main(String ar[])

        packEg p = new packEg();
        p.display();
    }
}
```

now we ill get output as "WELCOME TO SWARNANDHRA INSTITUE"

54. Interface:

An interface is a special case of abstract class, which contains all the abstract methods (methods without their implementation). An interface specifies what a class must do, but not how to do.

Using the keyword interface, we can fully abstract a class interface from its implementation. Interfaces are syntactically similar to classes, but they lack instance variable, and their methods are declared without anybody. Once it is defined, any number of classes can implement an interface. Also, once class can implement any number of interfaces.

55. Initializing interface:

An interface is defined much like a class. This is the general form of an interface:

```
access interface interface_name
{
        type final_varname1=value;
        type final_varname2=value;
        . . . .
        returntype method-name1(parameter_list);
        returntype method-name2(parameter_list);
        . . . .
}
```

here , access is either public or not used. When no access specifier is included, then defult access results, and the interface is only available to other members of the class package in which it is declared. when it is declared as public, the interface can be used by any other code. interface_name can be any valid identifier.

Methods which are declared have no bodies. they end with a semicolon after the parameter list. they are explicitly abstract methods.

variables are implicitly final and static, meaning they cannot be changed by the implementing class. They must be initialized with a constant value. All the methods and variable are implicitly public if the interface, itself is declared as public.

Eg:

```
interface
{
        int  pcode=999;                 // final variable
        String pname="HardDisk";        // final variable
        void show();                    // abstract method
}
```

56. Implementing interfaces:

Once an interface has been defined, one or more lases can implement that interface. To implement an interface, include the implementes clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this:

```
access class classname [extends superclass] [implements interface [,interface…]]
{
        // class body
}
```

Eg:

```
interface product
{
```

```
        int pcode=999;
        String pname="HardDisk";
        abstract void showProduct();
}
class ProductDesc implements Product
{
        public void showProduct()
        {
                System.out.println("Pcode="+pcode);
                System.out.println("PName="+pname);
        }
}
class DemoInterface
{
        public static void main(String ar[])
        {
                ProductDesc ob=new ProductDesc();
                ob.showProduct();
        }
}
```

Note:

- At the time of overriding, the interface methods must be declared as public.
- Interfaces are introduced to implement multiple inheritance indirectly in java

57. Variables in interface:

Interfaces can also be used to declare a set of constants that can be used in different classes. The constant values will be available to any class that implements the interface. The values can be used in any method, as part of any variable declaration, or anywhere where we can use a final value.

Example:

```
        interface A
        {
                int m=10, n=50;
        }
        class B implements A
        {
                void display()
                {
```

```
                System.out.println("m="+m+" , n="+n);
            }
        }
```

58. Extending interfaces

Like classes, interfaces can also be extended. That is , an interface can be subinterfaced from other interfaces. The new subinterface will inherit all the members of the superinterface in the manner similar to subclasses. This is achieved using the keyword extends as shown below

```
interface name2 extends name1
{
        body of name2;
}
interface ItemConstants
{
        int code=1001;
        String name="Fan";
}
interface Item extends ItemConstants
{
        void display();
}
class display implements Item
{
public void display()
{
        System.out.println(code+":"+name);
}
}
```

59. Applying interfaces

Implementing Multiple Inheritance:

A class can extend only class but it can implement multiple interfaces. This is one of the ways to ruse multiple inheritance in java. The class must override all the methods of all the interfaces.

eg:

```
interface citizen
{
```

```
        String name=”Rama”;
        abstract void show();
}
interface Employ
{
        int eno=65, salary=20000;
        abstract void showEmploy();
}

class professor implements Citizen, Employ
{
        int public=100;          //publications
        public void show()
        {
                System.out.print(“Name=”+name);
        }
        public void showEmploy()
        {
                System.out.println(“eno=”+eno+”salary=”+salary);
        }
}

class DemoMultiple
{
        public static void main(String ar[])
        {
                Professor p=new Professor();
                p.showCitizen();
                p.showEmploy();
                p.showProfessor();
        }
}
```

60. Differences between classes and interfaces

Both classes and interfaces contain methods and variables but with major difference. An

interface defines only abstract methods and final fields i.e., they don't specify any code for implementing these methods and data fields contain only constants. Using an interface we specify what a class must do, but not how to do. In contrast to interfaces, classes have instance variables and their methods have body. Once an interface is defined any number of classes can implement it. In some situations one class implements several interfaces.

since an interface defines only abstract methods, it is the responsibility of the class that implements an interface to define the code for implementation of these methods. Both classes and interfaces can be executed. Interfaces are in a different hierarchy from classes and because of this it is possible for classes which are unrelated in terms of the class hierarchy to implement the same interface.

The frequently used interfaces in different packages are as follows

package	**interface**
java.lang	Runnable, Cloneable
java.util	Enumeration, Observer
java.io	DataInput, DataOutput
java.awt	LayoutManager, MenuContainer
java.applet	AudioClip, AppletStub

61. StringTokenizer Class :

The processing of text often consists of parsing a formatted input string. Parsing is the division of text into a set of discrete parts, or tokens, which in a certain sequence can convey a semantic meaning. The StringTokenizer class provides the first step in this parsing process. We can enumerate the individual tokens contained in it using StringTokenizer .

To use StringTokenizer, you specify an input string and a string that contains delimiters. Delimiters are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter for example, ", ; :" sets the delimiters to a comma, semicolon, and colon. The default set of delimiters consists of the whitespace characters: space, tab, newline etc

The StringTokenizer constructors are shown here:

StringTokenizer(String str)
StringTokenizer(String str, String delimiters)
StringTokenizer(String str, String delimiters, Boolean delimAsToken)

In all versions, str is the string that will be tokenized. In the first version, the default delimiters are used. In the second and third versions, delimiters is a string that specifies the delimiters. Delimiters are not returned as tokens by the first two forms. Once you have created a StringTokenizer object, the nextToken( ) method is used to extract consecutive tokens. The hasMoreTokens( ) method returns true while there are more tokens to be extracted. Since StringTokenizer implements Enumeration , the hasMoreElements( ) and nextElement( ) methods are also implemented, and they act the same as hasMoreTokens( ) and nextToken( ) , respectively. The StringTokenizer methods are shown in below.

Method	Description
int countTokens( )	Using the current set of delimiters, the method determines the number of tokens left to be parsed and returns the result.
boolean hasMoreElements( )	Returns true if one or more tokens remain in the string and returns false if there are none.
boolean hasMoreTokens( )	Returns true if one or more tokens remain in the string and returns false if there are none.
Object nextElement( )	Returns the next token as an Object.
String nextToken( )	Returns the next token as a String.
String nextToken(String delimiters)	Returns the next token as a String and sets the delimiters string to that specified by delimiters

EXCEPTION HANDLING & MULTITHREADING

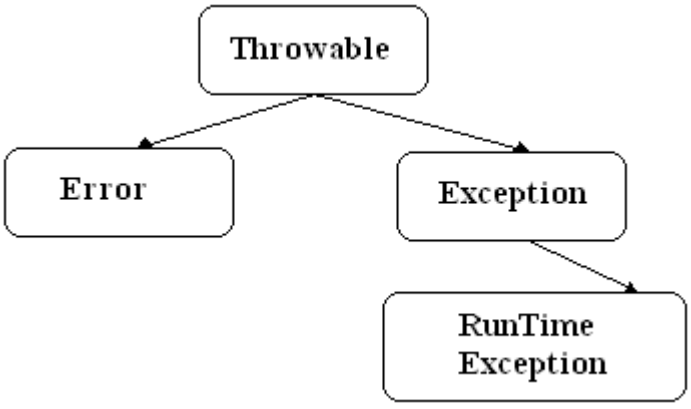
62. Exception: An Exception is defined as "an abnormal error condition that arises during our program execution"

When a Exception occurs in a program, the java interpreter creates an exception object and throws it out as java exceptions, which are implemented as objects of exception class. This class is defined in java.lang package

An Exception object contains datamembers that will store the exact information about the runtime error (Exception) that has occurred.

63. Exception Hierarchy:

All exception types are subclasses of the built-in class Throwable. Thus, Throwable is at the top of the exception class hierarchy.



'Exception' Class:

This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception, called Runtime Exception. Exceptions of this type are automatically defined for the programs that you write and include things such as division by

zero and invalid array indexing.

'Error' Class:

Which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the java run-time environment, itself. Stack overflow is an example of such an error.

Uncaught Exceptions:

This small program includes an expression that intentionally causes divide-by-zero error.

```
class Ex
{
        public static void main(String ar[])
        {
         int a=0;
        int b=10/a;
        }
}
```

Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

-----Here is the output generated when this example is executed------

javja.lang.ArithemeticException:/by zero

Java uses 5 keywords in order to handle the exceptions

1. try
2. catch
3. finally
4. throw
5. throws

try block:

The statements that produces exception are identified in the program and the statements are placed in try block

Syntax:

```
 try
{
        //Statements that causes Exception
}
```

catch block

The catch block is used to process the exception raised. The catch block is placed immediately after the try block.

Syntax:

```
catch(ExceptionType ex_ob)
```

```
{
        //Statements that handle Exception
}
```

finally block:

finally creates a block of code that will be executed after a try/catch block has completed. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.

Syntax:

```
finally
{
        // statements that executed before try/catch
}
```

throw:

It is possible to create a program that throws an exception explicitly, using the "throw " statement.

Syntax: throw throwable_instance;

Here throwable_instance must be an object type of Throwable class or subclass of Throwable. There are two ways to obtain a Throwable objects

1. using parameter into a catch clause
2. Crating one with the new operator

throws:

If a method is capable of causing an exception that it doen't handle, it must specify the behaviour to the callers of the method can guard themselves against that ex ce ptin. This can be don't by throws statement in the methods declarations.

A **throws** lists the types of exceptions that a method might throw.

Syntax: return_ type method_name(parameter-list)throws exception-list

```
        {
                //method body
        }
```

Here, exception-list is a comma-separated list of the exception that a method can throw.

64. User Defined Exceptions

It is possible to create our own exception types to handle situations specific to our application. Such exceptions are called User-defined Exceptions. User defined exceptions are created by extending **Exception** class. The **throw** and **throws** keywords are used while implementing user-defined exceptions.

Common Example for try , catch, throw , throws, finally and Userdefined exception:

```
import java.io.*;
class MyException extends Exception
```

```
{
        MyException(String msg)
        {
                super(msg);
        }
}


Class Test
{
        public static void main(String ar[])throws IOException
        {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter marks");
                try
                {
                        int marks=Integer.parseInt(br.readLine());
                        if(marks>100)
                        {
                                throw new MyException("Greater than 100");
                        }
                        System.out.println("Marks="+marks);
                }
                catch(MyException e)
                {
                        System.out.println(e.getMessage());
                }
                finally
                {
                        System.out.println("completed");
                }
        }
}
```

output-1:

```
Enter marks
99
Marks=99
completed
```

output-2:

```
Enter marks
101
Greater than 100
```

completed

65. Nested Try

A try block is placed inside the block of another try block is termed as Nested try block statements. If any error statement is in outer try block , it goes to the corresponding outer catch block. If any error statement is in inner try block first go to the inner catch block. If it is not the corresponding exception next goes to the outer catch, which is also not corresponding exception then terminated.

Example:

```
class NestedTry
{
        public static void main(String ar[])
        {
                try
                {
                        int a=args.length;//'a' stores no of command line args
                        int b=42/a;        //if a=0 it is Arithemetic Exception
                        try
                        {
                         if(a==1)
                                a=a/(a-a);

                         if(a==2)
                                {
                                 int c[]={3};
                                 c[20]=40;
                                }
                        }// end of inner try
                        catch(ArrayIndexOutOfBoundsException e)
                        {
                                System.out.println("Array Index exceeds");
                        }
                }//end of outer try
                catch(ArithmeticException e)
                {
                        System.out.println("Arithmetic Exception");
                }
        }
}
```

66. Multiple Catch Statements:

Multiple catch statements handle the situation where more than one exception could be raised

by a single piece of code. In such situations specify two or more catch blocks, each specify different type of exception.
Example:

```
class MultiCatch
{
	public static void main(String args[])
	{
		try
		{
			int a=args.length;
			int b=42/a;
			int c[]={3};
			c[20]=40;
		}
		catch(ArithmeticException e)
		{
			System.out.println(e.getMessage());
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println(e.getMessage());
		}
	}
}
```

67. **Thread** is a sequence of instsructions that is executed to define a unique flow of control. It is the smallest unit of code.

Creating Threads:

Threads are implemented in the form of objects that contain a method called **run().** The **run()** method is the heart and soul of any thread.

```
public void run()
{
…….
…….(statements for implementing thread)
…….
}
```

the run() method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called **start().**

A new thread can be created in two ways.

1. By extending Thread class: Define a class that extends **Thread** class and override its run() method with the code required by the thread.
2. By implementing Runnable interface: Define a class that implements Runnable interface. The Runnable interface has only one method, run(), that is to be defined in

the method with the code to be executed by the thread.

Extending Thread Class

We can make our class runnable as thread by extending the class java.lang.Thread. This gives us access to all the thread methods directly. It includes the following steps

1. Declare the class as extending the Thread class
2. Implement the run() method that is responsible for executing the sequence of code that the thread will execute.
3. Create a thread object and call the start() method to initiate the thread execution.

Example:

```
class A extends Thread
{
public void run()
{
        for(int i=1;i<=5;i++)
        {
                System.out.println("From Thread A : i "+i);
        }
}
}

class B extends Thread
{
public void run()
{
        for(int j=1;j<=5;j++)
        {
                System.out.println("From Thread B : j "+j);
        }
}
}
class ThreadTest
{
  public static void main(String ar[])
  {
        A a=new A();
        B b=new B();
        a.start();
        b.start();
   }
}
```

output-1:

```
From Thread A : i 1
```

From Thread A : i 2
From Thread B : j 1
From Thread B : j 2
From Thread A : i 3
From Thread A : i 4
From Thread B : j 3
From Thread A : i 5
From Thread B : i 4
From Thread B : i 5
output-2:
From Thread A : i 1
From Thread B : j 1
From Thread B : j 2
From Thread B : j 3
From Thread A : i 2
From Thread B : j 4
From Thread B : j 5
From Thread A : i 3
From Thread A : i 4
From Thread A : i 5

68. Implementing Runnable Interface:

The Runnable interface declares the run() method that is required for implementing threads in our programs. To do this, we must perform the steps listed below:

1. Declare the class as implementing the Runnable interface.
2. Implement the run() method
3. call the thread's start() method to run the thread.

Example:

```
class A implements Runnable
{
        public void run()
        {
                for(int i=1;i<=5;i++)
                {
                        System.out.println("Thread A: i= "+i);
                }
        }
}
class RunnableTest
{
        public static void  main(String ar[])
        {
```

```
            A ob=new A();// A's object 'ob'
            Thread t=new Thread(ob);
            t.start();
        }
}
```

output:

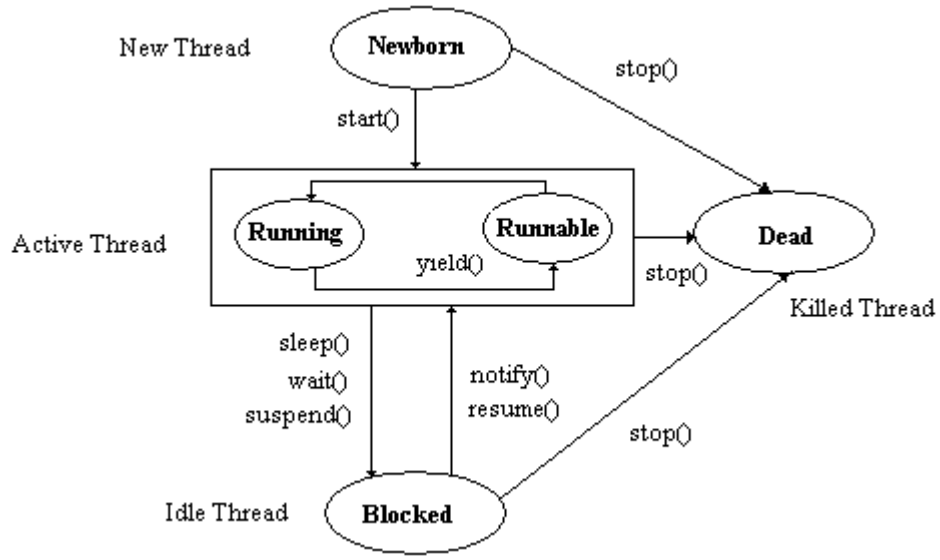
```
Thread A: i= 1
Thread A: i= 2
Thread A: i= 3
Thread A: i= 4
Thread A: i= 5
```

69. Lifecycle of a Thread

During the lifetime of a thread, there are many states it can enter. They include:

1. NewBorn State
2. Runnable State
3. Running State
4. Blocked State
5. Dead State

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in below figure.



NewBorn State:

When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running. At t his state, we can do only one of the following things with it:

- Schedule it for running using start() method
- Kill it using stop() method

Runnable State:
The runnable state means that the thread is ready for execution and is waiting for the availability of the processor. If we want a thread to relinquish control to another thread to equal priority before its turn comes, we can do so by using the **yield()**

Running State:
Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread.

Blocked State:
A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running s tate. This happens when the thread is suspended, sleping, or waiting in order to satisfy certain requirements. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

Dead State:
Every thread has a lifecycle. A running thread ends its life when it has completed executing its run() method. It is natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it. It is done by stop() method.

Synchronizing Threads:
Synchronization of threads ensures that if two or more threads need to access a shared resource then that resource is used by only one thread at a time. You can synchronize your code using the synchronized keyword. You can invoke only one synchronized method for an object at any given time.
Synchronization is based on the concept of monitor. A monitor, also known as a semaphore, is an object that is used as a mutually exclusive lock. All objects and classes are associated with a monitor and only one thread can won a monitor at a given time.
The monitor controls the way in which synchronized methods access an object or class. When a thread acquires a lock, it is said to have entered the monitor. The monitor ensures that only one thread has access to the resources at any given time. To enter an object's monitor, you need to call a synchronized method.

When a thread is with in a synchronized method, all the other threads that try to call it on the same instance have to wait. During the execution of a synchronized method, the object is locked so that no other synchronized method can be invoked. The monitor is automatically released when the method completes its execution. The monitor can also be released when the synchronized method executes the wait() method. When a thread calls the wait() method, it temporarily releases the locks that it holds.
example: See the example which I was given (u can find it in observation or in record or in notes)

70. The Synchronized Statement:

Synchronization among threads is achieved by using synchronized statements. The synchronized statements is used where the synchronization methods are not used in a class and you do not have access to the source code. You can synchronize the access to an object of this class by placing the calls to the methods defined by it inside a synchronized block .
syntax:

```
synchronized(obj)
{
        // statements;
}
```

71. Inter-thread communication

Java supports inter-thread communication using wait(), notify(), notifyAll() methods. These methods are implemented as final methods in **Object**. So all classes have them. all three methods can be called only from within a synchronized context.

- **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()**
- **notify()** wakes up the first thread that called **wait()** on the same object.
- **notifyAll()** wakes up all the threads that called **wait()** on the same object. The highest priority thread will run first.

72. Daemon Threads:

A daemon thread is a thread that runs in the background of a program and provides services to other threads. The daemon threads are background service provides for other threads. For example, Garbage collector, clock handler, and screen updater are the daemon threads. By default no thread that you create is a daemon thread. The setDaemon() method is used to convert a thread to a daemon thread.
Eg:
Thread1.**setDaemon**(true);
setDaemon() method is used to determine if a thread is a daemon thread.
Boolean b=Thread1.**isDaemon();**
'b' contains true if Thread1 is daemon, otherwise stores false.

73. ThreadGroup

ThreadGroup creates a group of threads. It defines these two constructors

```
ThreadGroup(String groupname);
ThreadGroup(ThreadGroup parentob,String groupname);
```

For both forms, group name specifies the name of the thread group. Thread groups offer a convenient way to manage groups of threads as a unit.
For example, imagine a program in which one set of threads is used for printing a document, another set is used to display the document on the screen, and another set saves the document to a disk file. If printing is aborted, you will want an easy way to stop all threads related to printing.

74. Thread Priorities:

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower priority threads.

Java assigns to each thread a priority that determines how that thread should be treated with respected to the others. Thread priorities are integers that specify the relative priority of one thread to another. Thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch. The rules that determine when a context switch takes place.

- A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping, or blocking on pending I/O. in this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given to the CPU.
- A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted no matter what it is doing by a higher-priority thread.

In cases when two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as windows, threads of equal priority are time sliced automatically in round-robin fasion.

To set a thread's priority, use the setPriority() method, which is a member of Thread. This is its general form:

final void setPriority(int level)

here, level specifies the new priority setting for the calling thread. The value of level must be with in the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10 respectively. To return a thread to a default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as final variables within Thread.

We can obtain the current priority setting by calling the getPriority() method of Thread

its general form -> final int getPriority();

Networking

75. Basics of network programming

76. Network: More than one computer systems are connected to sent and receive the information with each other is considered as a Network. due to the networking there is a communication between the systems. So, that they are shared data as well as resources.

77. Internet: An interconnection of networks is considered as a internet or intranet(**L**ocal **A**rea **N**etwork). It is the name for vast of worldwide systems, which consists of different types of networks or different servers. The computers that are connected to the Internet is considered as a Host.

78. Socket Overview:

Networks sockets are used for networking the systems. A Network socket is like an electrical socket at which various plugs around the network has a standard way of delivering their

payload. For transmission the data, the sockets are used some protocols.

79. **Protocol** is a standard set of rules, which is used to transfer the data from one system to another. Protocols are classified into two types as

Connection oriented protocols: Eg: TCP/IP Protocols
Connection-less Protocols Eg: UDP Protocols

Internet Protocol(IP) is a low-level routing protocol that breaks data into small packets and sends them to an address across the network. It is not guarantee to deliver the packets to destination.

Transmission Control Protocol(TCP) is a higher level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably tranmit the data.

User Datagram Protocol(UDP) used directly to support fast, connectionless, unreliable transfer of packets.

Addresses:

When there are tens of millions of computers that are connected in a single network, an addressing scheme can be used to select an application that is running on any one of those multiple computers. The computers connected with in an internet have addresses called IP addresses. This address has the form

Num1.Num2.Num3.Num4

There are four numbers separated with dots and each number can have any value between 0 and 255

For example: 192.27.168.35

Another way of designating a particular computer is through domain name address. The domain names are similar to that of IP addresses as they are written as a sequence of items that are separated by periods(dots).Domain names point to a particular machine through multiple levels. The levels on the right are more general and as they are read to the left, they become more specific.

Example. www.jntu.edu.in

Port: A Location where the clients and servers can exchange information is called a port. Ports can be designated using integer numbers. The numbers that are less than 1024 are reserved for predefined services such as file transfer, email etc. So, a user defined port can have an integer value that is greater than 1024.

Simple client server program

program on Server side:

```
import java.net.*;
class server
{
public static void main(String arg[])throws Exception
```

```
{
        System.out.println("Server Activated");
        InetAddress a=InetAddress.getLocalHost();
        Socket s=new Socket(a,1080);// or u can put “localhost” instead of Inetaddress ‘a’
}
}
```

program on client side:

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

class client
{
public static void main(String arg[])throws Exception
{

        System.out.println("Client Activated");

        ServerSocket ss=new ServerSocket(1080);
        Socket s=ss.accept();
        Date d=new Date();
        System.out.println(d.toString());
}
}
```

//Write output & explanation in your words.

Multiple clients:

//Write above server/client program. But in case client programs u need write same client program for few times….. i.e

```
class client1
{
public static void main(String arg[])throws Exception
{

        System.out.println("Client1 Activated");

        ServerSocket ss=new ServerSocket(1080);
        Socket s=ss.accept();
        Date d=new Date();
        System.out.println(“date at client 1”+d.toString());
}
}
class client2
{
public static void main(String arg[])throws Exception
```

```
{
        System.out.println("Client2 Activated");
        ServerSocket ss=new ServerSocket(1080);
        Socket s=ss.accept();
        Date d=new Date();
        System.out.println(“date at client 2”+d.toString());
}
}
//U must write server side program too
// Refer Running notes too
```