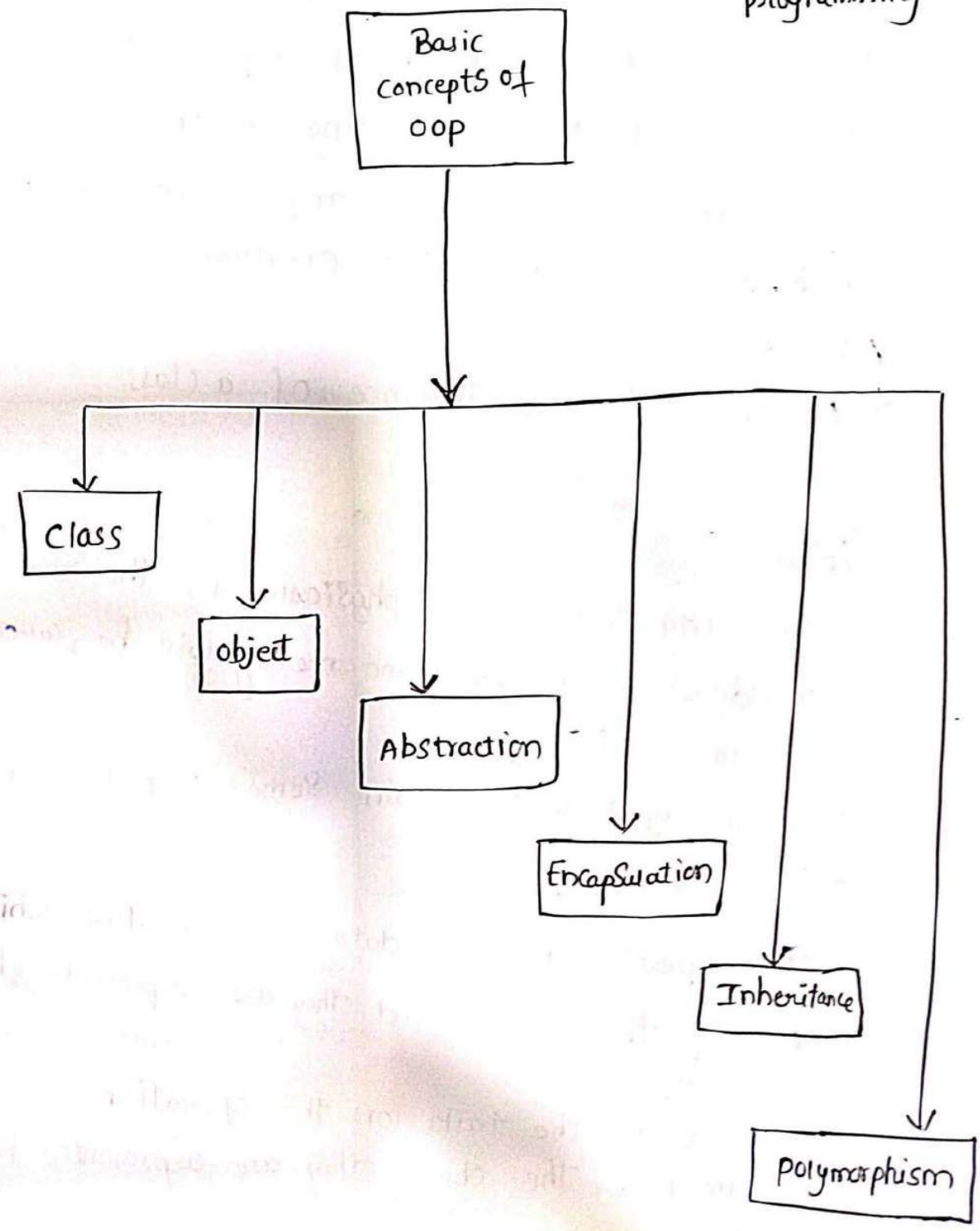
/* object oriented programming :

Basic concepts, principles :

oop - object oriented programming



(1) Class :

→ A class is a collection of common properties and common Actions of a group of objects.

→ A class can be considered as a plan (or) a model (or) a blue print for creating an object.

→ For a class we can create any number of objects.

→ With out class the object creation is not possible.

→ An object is an instance of a class.

(2) Object :

→ An Entity that exists physically in the real world which requires some memory will be called as an object.

→ Every object will contain some "properties" and some "Actions".

→ The properties are the data (or) information which describes the object and they are represented by variables.

→ Actions are the tasks (or) the operations performed by the object they are represented by methods.

(3) Encapsulation:

→ Encapsulation is a process of binding the variables and methods into a single Entity.

(or)

→ Wrapping / Binding up of variables (data) and methods (code) into a single unit.

(4) Abstraction:

→ Representing Essential features without including background details.

(or)

→ Abstraction in java refers to hiding the implementation details of a code and exposing only the necessary information to the user.

(5) Inheritance:

→ Inheritance is a process of acquiring the members from one class to another class.

→ using this we can achieve reusability and there by reduce the code size and reduce the code size and reduce the development time.

(6) Polymorphism:

→ If a single entity shows mutiple behaviours then it is called as "polymorphism".

→ Ex: Method overloading and Method overriding.

* Java Introduction:

→ Java is Highlevel, general purpose, object oriented programming language developed by Sun microSystem in 1995.

→ It was invented in June 1991 but the first version java is officially released in 1996 by Sun Micro System.

→ currently it is being maintained by oracle co-oportion.

→ It is to write a program on multiple OS.

→ It runs on different platforms.

Ex: Windows, Mac OS, unix

→ It is also called WORA.

write once, run anywhere.

→ Before Java we are using C, C++....

→ Java is aset of features of C & C++.

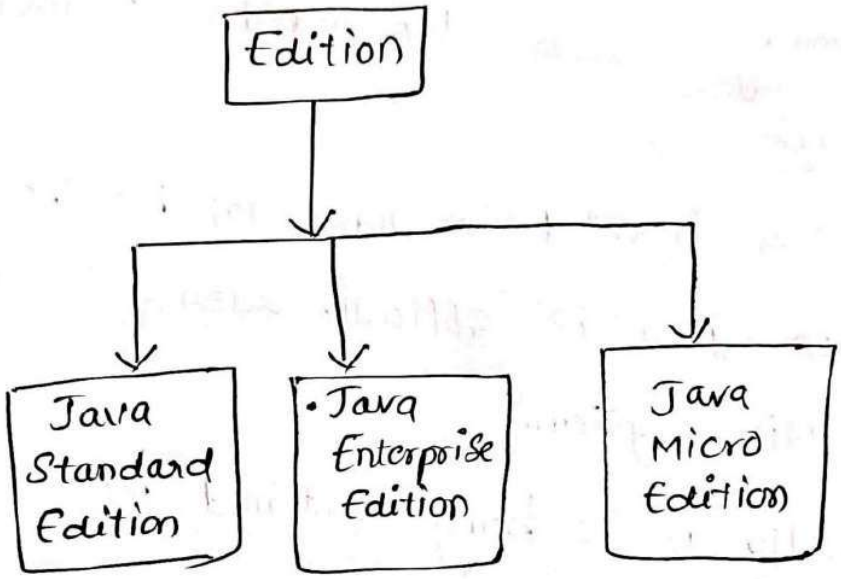
Java = C (Syntax's) + C++ (OOP Concepts) + Additional feature

Java → Platform Independent

C + C++ → System Dependent Language

Edition Of JAVA:

— Java comes in three editions



Java Standard Edition (SE):

→ Develop applications that run on desktop.

Java Enterprise Edition (EE):

→ Develop Server Side Application.

Java Micro Edition:

→ Develop applications for mobile Application.

JDK (Java Development Kit)

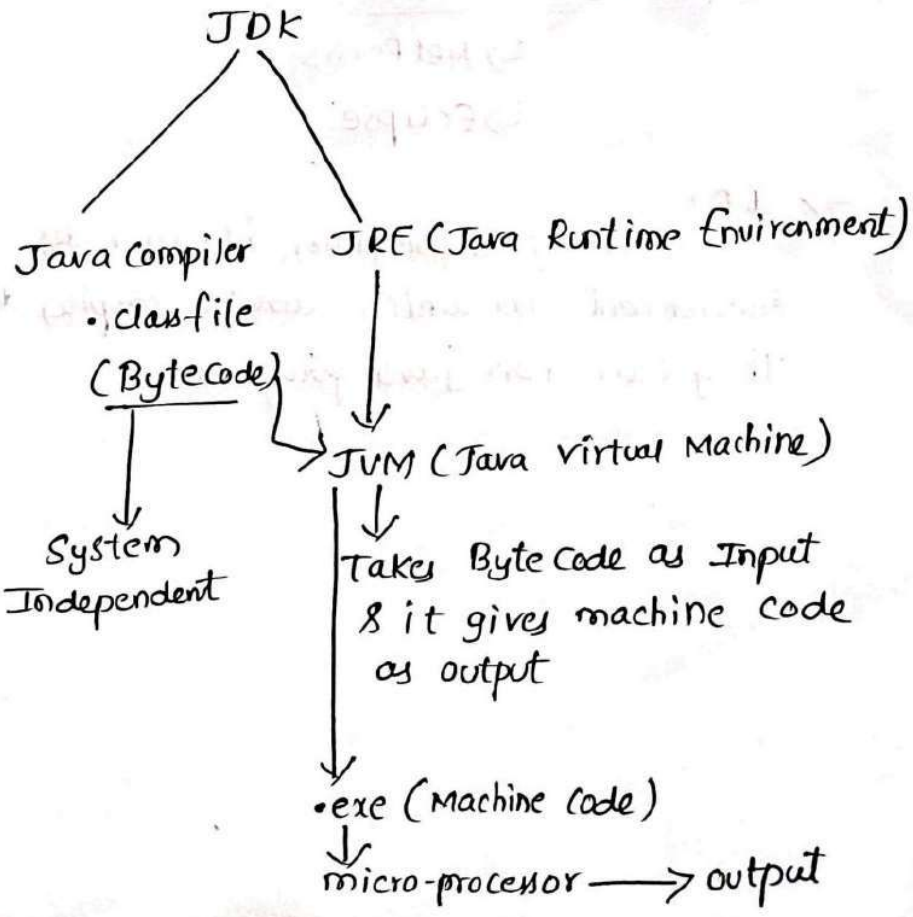
→ Set of programs that enable us to develop our programs.

→ Contain JRE (Java Runtime Environment) that is used to run our programs.

→ JRE & JDK contain JVM (Java virtual Machine)

→ JVM executes our java programs on different machines.

→ java is independent.



IDE (Integrated development environment)

→ A program that allows us to:

- write - Source code
- compile - machine code
- Debug - tools to find errors
- Build - file that executed by JVM.
- Run - execute our program

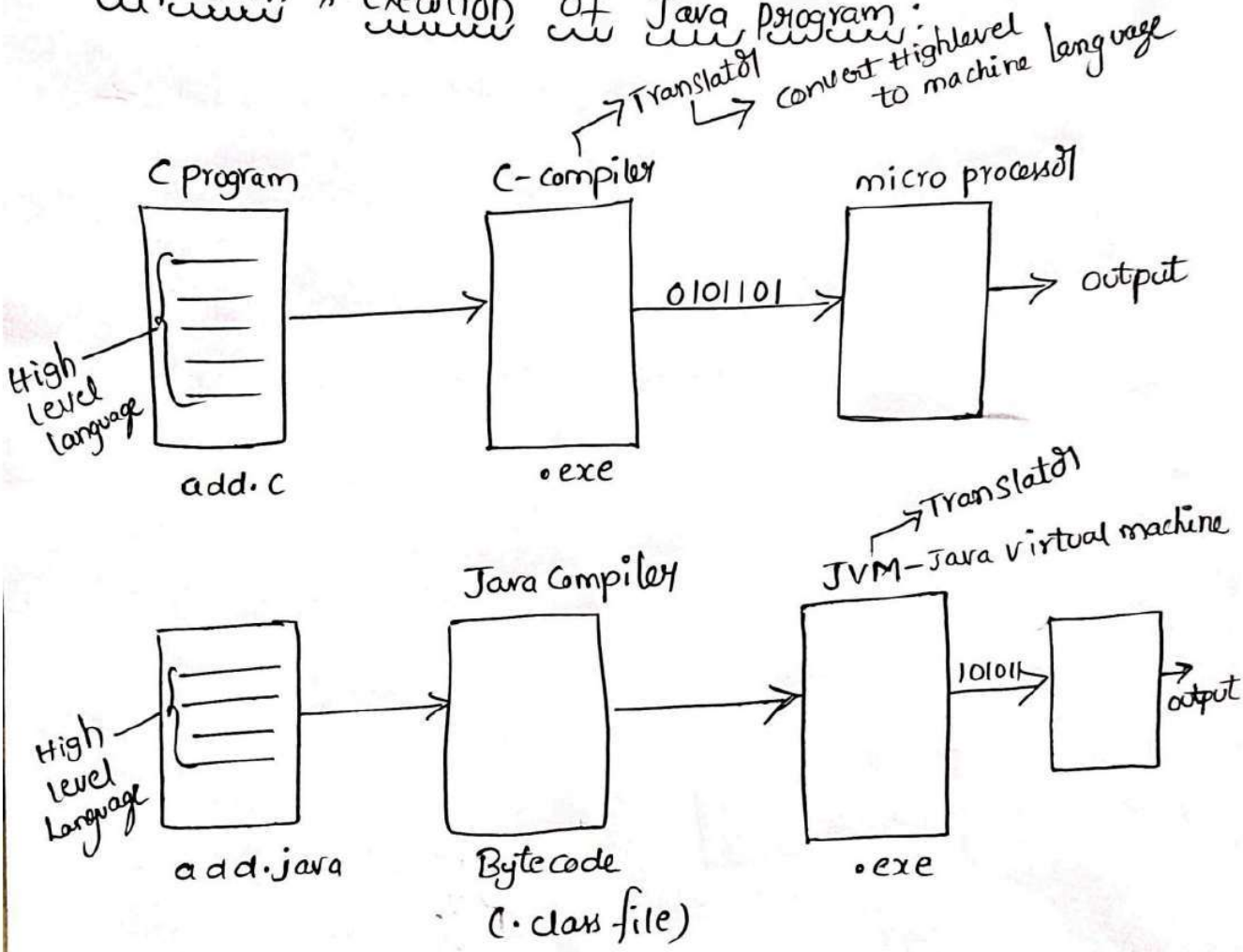
→ Development is faster and Easier.

→ popular Java IDE's:

- NetBeans
- Eclipse

→ IDE
↳ It is a program, it provides complete Environment in which write, compile, debug and they can run java program.

* Compilation & Execution of Java program:



* Java Feature / Buzzwords:

1. Object oriented: (Abstraction, Encapsulation, Inheritance, polymorphism)

→ object oriented throughout - no coding outside of class definations, including main().

2. platform Independent:-

→ Java works on different platforms (Windows, Linux, Mac etc.) and programs developed in JAVA are executed anywhere on any System.

3. Simple:

→ They made java Simple by eliminating difficult concept of pointers and java is simple because it has the Same Syntax of C and C++.

4. portable:

→ Java is portable because of its "write once, run anywhere" principle, which allows Java code to be compiled into platform independent bytecode that can be executed on any device with ajava virtual machine (JVM).

5. Robust: because of the following concept included in Java it is called as robust.

1. Exception Handling
2. Storage Management
3. Automatic Garbage Collection.

6. Architectural Neutral / Neutral:

→ Java is considered architecture Neutral because it can run on any platform (or) OS without requiring the code to be modified.

Ex:

C

int ⟶ 16 bit — 2 bytes of Memory
int ⟶ 32 bit — 4 bytes of Memory

Java

int ⟶ 16 bit / 32 bit } — 4 bytes of Memory.

→ Here Architecture is fixed.

7. Interpreted :

→ Java can be considered both a compiled and an interpreted language because its source code is first compiled into a binary byte code.

→ This byte code runs on the Java Virtual Machine, which is usually a Software-based interpreter.

8. High performance :

→ The Java language supports many high-performance features such as multithreading, just-in-time compiling.

9. Multi Thread :

Threading : A thread represents an individual process to execute a group of statements. JVM uses several threads to execute different blocks of code.

Multi Thread – more than one thread run at time. (or) parallel (or) Simultaneously executed.

10) Distributed:

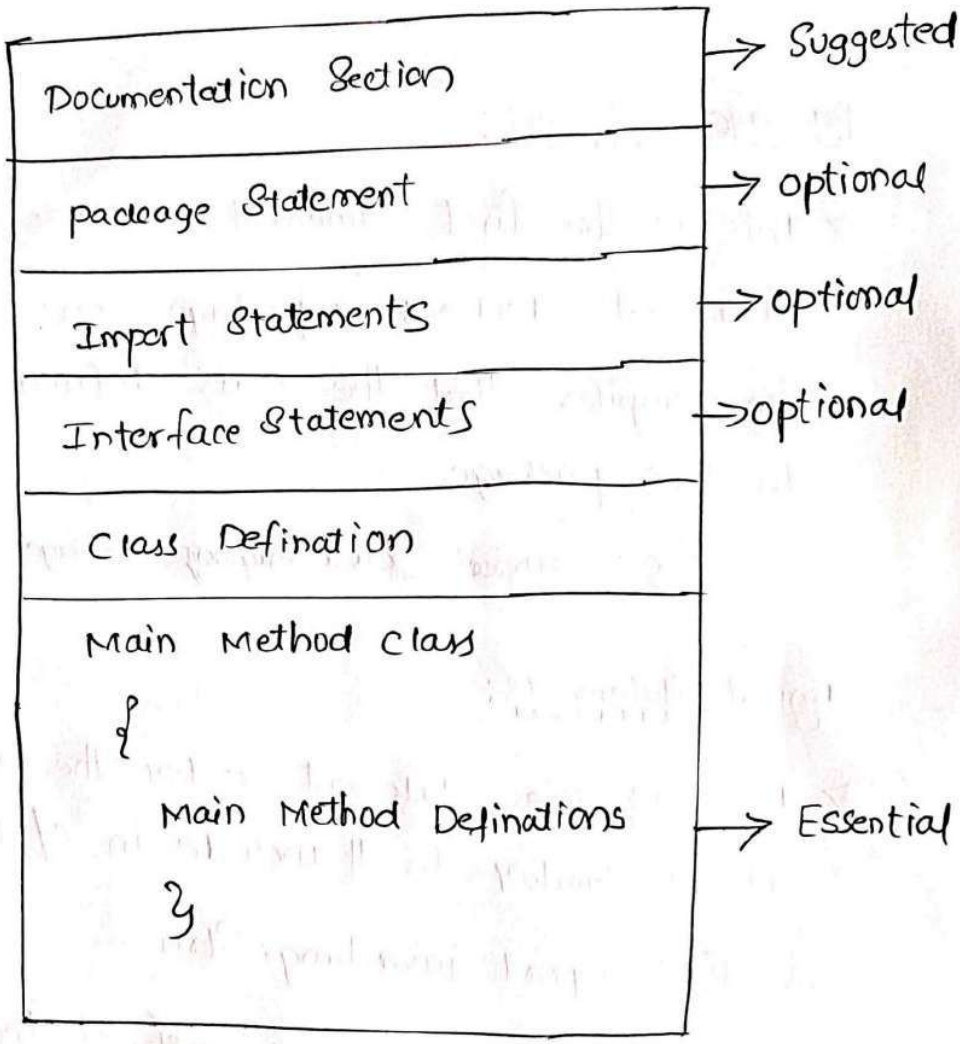
→ It uses java, a high level programming language, to distribute data & tasks across multiple computers.

→ RMI (Remote method Invocation) and EJB (Enterprise Java Bean) are used for creating distributed applications. This feature of Java make us able to access files by calling the methods from any m/c on the Internet.

11) Security:

→ Security problems like tampering and virus threats can be eliminated (or) maintained by using java on internet.

* Structure of Java program:



(1) Documentation Section:

→ It consists of comment lines (program name, author, date, ...)

//- for single line comment

/*----*/ - multiple line comments.

/**----
*----- - Known as documentation comment, which
----/
generated documentation automatically.

Package Statement:

→ This is the first statement in java file. This Statement declares a package name and informs the compiler that the class defined here belong to this package.

Ex: ~~import java.lang.~~ package Student

Import Statements:

→ This is the statement after the package Statement It is similar to # include in C/C++.

Ex: import java.lang.String

The above statement instructs the interpreter to load the String class from the lang package.

Note:

→ Import Statement should be before the class declarations.

→ A Java file contain N number of import statements.

Interface Statements:

→ An Interface is like a class but includes a group of method declarations.

Note: Methods in interfaces are not defined just declared.

Class Definitions:

→ Java is a true object oriented programming, So classes are primary and essential elements of java program. A program can have multiple class definitions.

Main method class:

→ Every Stand-alone program requires a main method as its starting point. As it is true OOP the main method is kept in a class definition.

→ Every Stand-alone small java program should Contain at least one class with main method defination.

Rules for writing JAVA programs:

1. Every Statement ends with a Semicolon.
2. Source filename and class name must be the same.
3. It is a case-sensitive language.
4. Every thing must be placed inside a class, this feature makes JAVA a true object oriented programming language.

* Simple Java program:

```
/* This is a Simple java program.
   program name: Sample.java */
class Sample
  {
    public static void main (String args[])
      {
       System.out.print (" Hello Java program");
      }
  }
```

Save: Sample.java

compile: javac Sample.java

Run: java Sample

output:

Hello Java program

The name of the Source file and name of the class name(which contains main method) must be the Same, because in JAVA all code must reside inside a class.

Class Sample:

→ The keyword class is used to declare a class and Sample is the JAVA identifier i.e name of the class.

public Static void main(String args[]):

public:

→ The keyword "public" is an access Specifier that declares the main method as unprotected therefore making it accessible to all other classes. The opposite of public is private.

Static:

→ The keyword Static allows main() to be called without creating any instance of that class.

void:

→ The keyword void tells the compiler that main() does not return any value.

main():

→ main() is the method called when a java application begins.

String args[]:

→ Any information that you pass to a method is received by variables specified with in the set of parenthesis that follow the name of the method.

→ These variables are called parameters. Here args[] is the name of the parameter of String type.

System.out.print:

→ It is Eqival to printf() (or) cout<<, since java is a true object oriented language, every method must be part of an object.

Compiling the program:

C:> javac Sample.java

Extension is compulsory at the time of compiling

javac - is a compiler, which create a file called Sample.class (Contains the Byte code)

Note:

→ When java Source code is compiled each class is put into a Separate.

.classfile

Executing the program:

C:> java Sample

java - is interpreter which accepts .class file name as a command line argument.

that's why the name of the Source code file and class name should be equal, which avoids the Confusion.

output:

Hello Java program

* Class and object:

Class:

→ The class keyword is used to declare a new java class, which is a collection of related variables and/or methods.

→ Classes are the basic building blocks of object-oriented programming.

→ A class is a template for an object.

→ class is a blueprint of an object.

Class Declaration:

```
class <class-name>
{
    return type variable1;
    return type variable2;        // Variables (or) Properties
        :
    return type method name1(parameters)   <- optional
    {
        ---;  // Statements
    }
    return type method name2
    {
        ---;  // Statements
    }
}
```

Methods (or) Actions

Object:

→ An object in java is a basic unit of object-oriented programming and represents real-life entities.

→ objects are the instances of a class.

→ object contains variables and methods.

→ object is physical entity.

Syntax:

<class-name> <object-name> = new <class-name>();

Example:

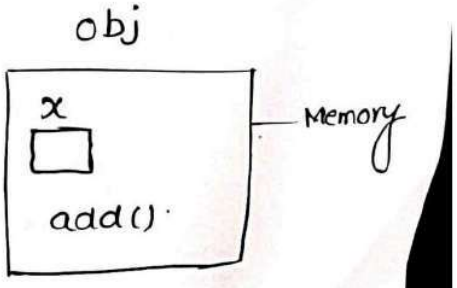
used to allocate memory for the object

Kalam obj = new Kalam();

↓

keyword

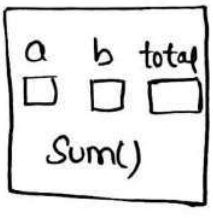
```
Class Kalam
{
  int x;
  void add();
  {
    ---; //stmts
  }
}
```



Example program:

```
import java.lang.*;
Class Addition
{
  int a, b, total; // Instance Variable
                   (or)
                   Class variable

  void Sum1()
  {
    a = 5;
    b = 10;
    total = a+b;
  System.out.print(" Addition of Two numbers = "+total);
  }
}
class Maddition
{
  public static void main (String args[])
  {
  Addition obj = new Addition();
       obj.sum();
  }
}
```



Save: Maddition.java

Compile: javac Maddition.java

Run: java Maddition

* Difference between class and object:

Class	object
→ class is a blueprint from which objects are created.	→ object is instance of the class.
→ class is a collection of objects.	→ object is a real world Entity such as book, pen, board, person, table, etc.
→ class is a logical Entity	→ object is a physical Entity.
→ class is defined only once.	→ objects can be created by many as per requirement.
→ class doesn't Allocate memory for variables when it is created.	→ objects Allocate memory for variables when it is created.
→ class is declared using class keyword.	→ new operator

Syntax:

```
Class <class-name>
{
  variables;
  method
}
```

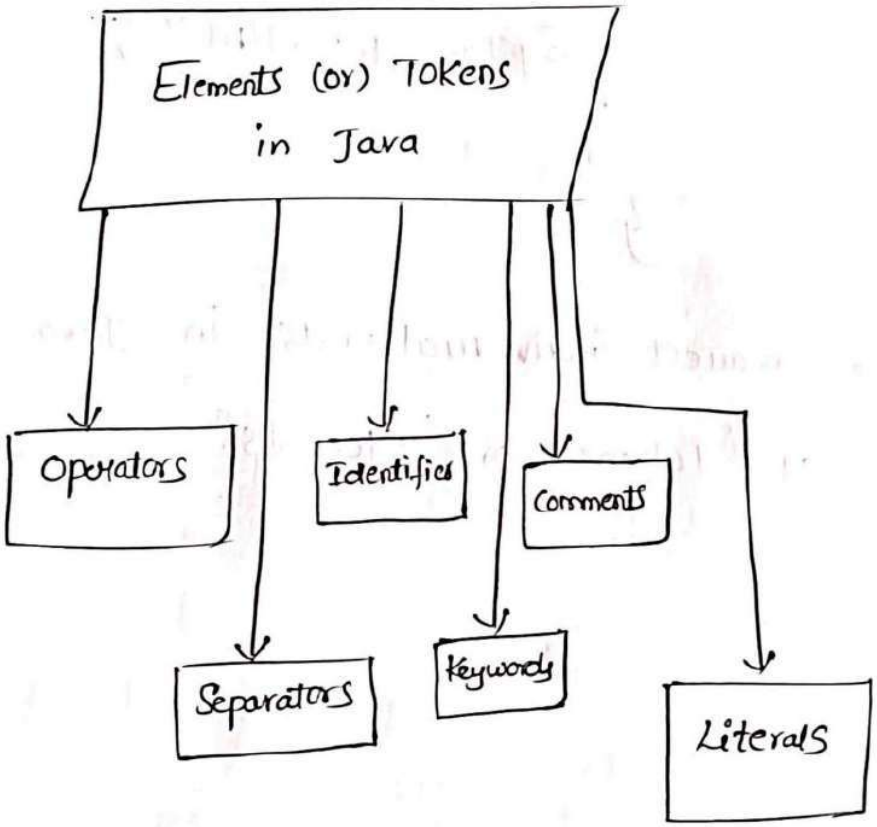
```
classname obj = new classname();
```

* Elements (or) Tokens in java Program:

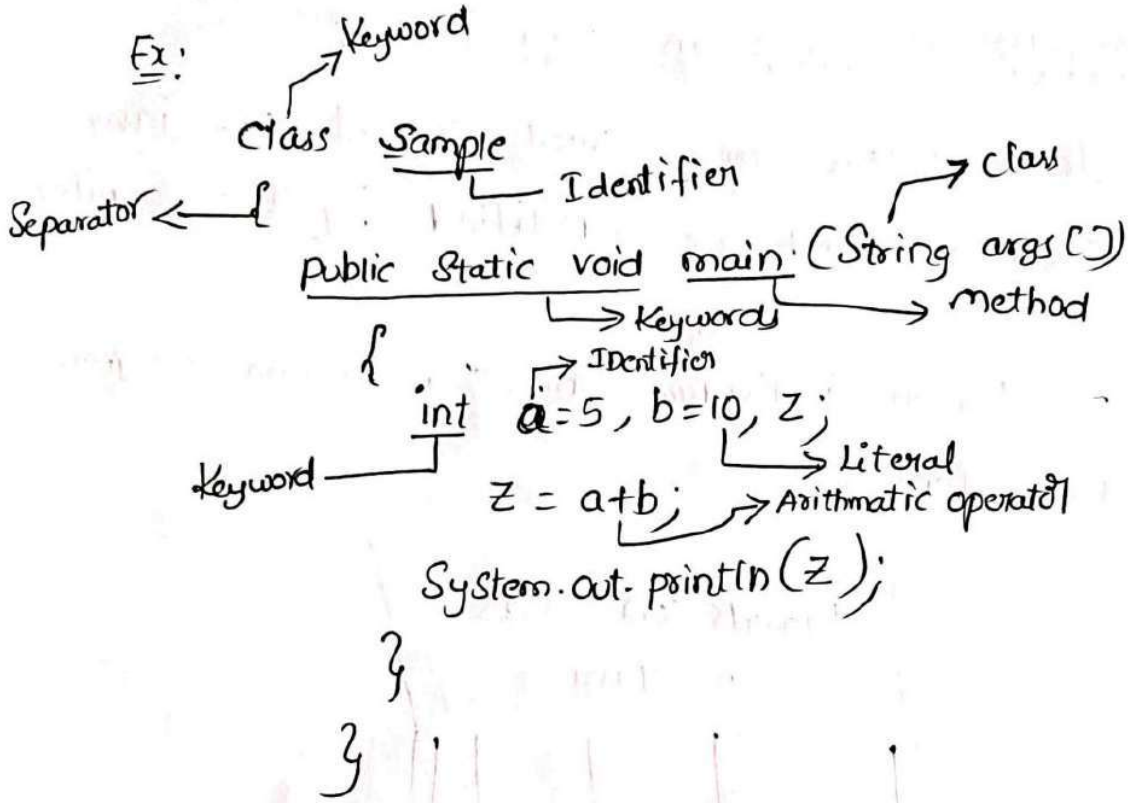
→ Java Tokens are Smallest element of java Program which are identified by the compiler.

(or)

Smallest individual units in a Java program as "Tokens".



Ex:



→ Smallest individual units in Java Program as "Tokens" (or) "Elements".

Keywords:

→ Keywords also known as resolved words (or) predefined words that have special meaning to compiler.

Ex: int x;

Compiler allocates memory for variable x.

→ all keywords must be used in lower case only And white space not allowed.

→ May not be used as identifier (Class names, variables and methods) names.

List of keyword in java:

primitive types and void	Access and non Access modifiers
1. Boolean	1. public
2. byte	2. private
3. char	3. protected
4. Short	4. Static
5. int	5. final
6. long	6. transient
7. float	7. abstract
8. double	8. volatile
9. void	9. Synchronized
	10. native

Declarations

1. class
2. interface
3. enum
4. extends
5. package
6. implements
7. throws

Control flow

1. if
2. else
3. Switch
4. default
5. case
6. for
7. while
8. do
9. break
10. return
11. finally
12. try
13. catch
14. Final
15. throw

Others

1. true
2. false
3. null
4. import
5. new
6. instanceof
7. Super
8. assert
9. Strictfp
10. Const
11. this.

Identifiers:

→ Identifiers are the names given to classes, variables, method etc.,

Ex: class Sample
{ ← Identifier
}

Ex: void add()
{ ← Identifier
}

Ex: int x;
← Identifier

Rules for Identifiers:-

1. Identifier can not be a keyword.

Ex: int y; ✓

int float; ✗

2. Identifiers are case sensitive i.e., upper case and lower case alphabets treated as different.

Ex: int A, a; // here 'A' and 'a' are treated as different variables.

3. It can have Sequence of letters and digits and must begin with a alphabet only (or) $, _ Symbols are allowed.
(_ ↓ UnderScore)

Ex: int a1; ✓ - valid
int 5n; X - Invalid
int_a; ✓ - valid
int $k; ✓ - valid

4. White Spaces are not allowed

Ex: int display; ✓ - valid
int d isplay; X - Invalid
(d ↓ Space)

5. Symbols like #, @ are not allowed.

Operators:

→ Operators is a Special Symbol which operates on operandy to produce results.

→ which is used for mathematical and logical calculations.

Ex: a + b (+ → Operator; a, b → Operandy)
30 + 40 (+ → Operator; 30, 40 → Operands)

Basic Arithmetic Operators:

→ Arithmetic operators used to perform common mathematical operations.

Operator	Symbol	Example a=21, b=5
Addition	+	a+b = 26
Substraction	-	a-b = 16
Multiplication	*	a*b = 105
Division	/	a/b = 4.2
Modulus	%	a%b = 1

% → Modulus operator
↓
it results remainder

Example Program:

```
Class ArithmeticExp
{
    public static void main (String args[])
    {
        int a=21, b=5;
        /* System.out.println (a+b); //output-26 */
        System.out.println ("Addition = " +(a+b));
        System.out.println ("Substraction = " +(a-b));
        System.out.println ("Multiplication = "+(a*b));
        System.out.println ("Modulus = " +(a%b));
        System.out.println ("Division = "+(a/b));
        /* System.out.println ("Division = "(double)a/b)); //type casting */
                                                    output: 4.2
    }
}
```

Output

Addition = 26

Substraction = 16

multiplication = 105

Division = 4

Modulus = 1

* Relational (or) Comparision

→ Relational operators are used to check the relationship between two operands.

Operator	Symbol	Example a=10, b=5	Example a=1, b=5
Greater than	>	a > b = true	a > b = false
Greater than Equal to	>=	a >= b = true	a >= b = false
Less than	<	a < b = false	a < b = true
Less than Equal to	<=	a <= b = false	a <= b = true
Not Equal to	!=	a != b = true	a != b = true
Equal to	==	a == b = false	a == b = false

Example program:

```
class RelationalExp
{
  public static void main(String args[])
  {
    int a=10, b=5;
    System.out.println(a>b);
    System.out.println(a>=b);
    System.out.println(a<b);
    System.out.println(a<=b);
    System.out.println(a!=b);
    System.out.println(a==b);
  }
}
```

output:

```
true
true
false
false
true
false
```

* Boolean logical operator:

→ The Logical && operator doesn't check Second condition if first condition is false. It checks Second condition only if first one is true.

→ The logical || operator doesn't check Second condition if first condition is true. It checks second condition only if first one is false.

Operator	Symbol	Example
Logical AND	&&	true && true = true true && false = false false && true = false false && false = false
Logical OR	||	true && true = true true && false = true false && true = true false && false = false
Logical NOT	!	!true = false !false = true

Example program

```
Class LogicalExp
{
  public static void main(String args[])
  {
    boolean A=true, B=false;
    S.o.pln(A&&A); //Logical AND
    S.o.pln(A&&B);
    S.o.pln(B&&A);
    S.o.ph(B&&B);
    S.o.pln(A||A); //Logical OR
    S.o.pln(A||B);
    S.o.pln(B||A);
    S.o.pln(B||B);
    S.o.pln(!A);
    S.o.pln(!B);
  }
}
```

Output:

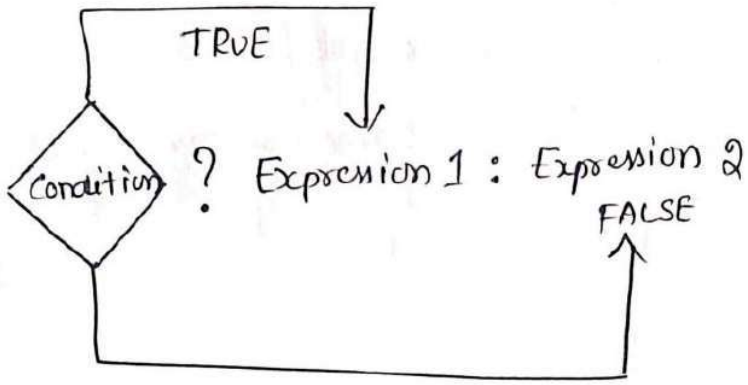
```
true      } Logical AND
false
false
false
false    } Logical NOT
true
true
true
false
```

* Ternary Operator:

→ A Ternary operator Evaluates the test condition and executes the Expression based on the result of the condition. Also called conditional operator.

Syntax:

Variable = condition ? Expression 1 : Expression 2;



Program:

```
class Ternaryfx
{
  psvm (String args[])
    {
      int a=12, b=5, large;
      if(a>b)
      {
        large=a;
      }else
      {
        large=b;
      }
System.out.print("Large = " + large);
    }
}
```

apply Ternary operator

output

Large = 12

Program: class TernaryExample

```
{
    psvm(String args[])
    {
        int a=12, b=5, large;

        large = (a>b) ? a : b;

        System.out.println("Large Number is = "+large);
    }
}
```

output

Large Number is = 12

* instanceof

→ An instanceof in Java is a comparsion operator which, given an object instance.

→ It checks whether that instance is of a specified type (class/ Sub-class ~~(interface)~~) or not.

→ Just like other compassion operators, it returns tru (or) false.

Example:

```
class TestInstanceof
{
    psvm (String args[])
    {
TestInstanceof obj = new TestInstanceof(); //Creating instance of a class
        if (obj instanceof TestInstanceof)
        {
        S.o.pln ("yes");
        } else { S.o.pln ("No");
        } } }
```

Output:

yes

* Separators in Java :-

→ A Separator is a symbol that is used to Separate a group of code from one another.

Parenthesis :→ () → used to enclose an argument in the method defination. Also used for defining defination. Also used for defining the expression in control Statement etc.,

Ex :

1) ob.add (5,10);

```
void add (int x, int y)
  {
  }
```

2) if (condition)

```
{
}
```

3) Switch (expression)

```
{

}
```

Curly braces :→ { } - used to define a block of code for classes and method. Also used to contain the values of arrays. etc.,

Ex: 1) class Kalam
{
}

2) void sum(int a, int b)
{
}

3) int arr[] = {1,2,3,4};

Brackets :→ [] - used to declare an array type. Also used when difference array values etc,

Ex: int arr[] = new int[10]; // 1-Dimensional array
(Subscript)

int arr[][] = new int[4][4]; // 2-Dimensional Array

Semicolon: → ; - used to Separate (or) terminate the Statement etc;

Ex:

```
for (k=1; k<=10; k++)
System.out.println ("welcome Java");
```

And many more

comma: → , - used to Separate identifier (or) variable declaration. Etc;

Ex: int x, y, z;

Period or dot: → . - to Separate package name from Sub-packages and Classes. Also used to Separate a variable

Ex: import java.util.Scanner;

obj.add();

NOTE:

→ Semi colon(;), Comma(,), and dot(.). Are also called punctuators

* Increment (++) and Decrement (--) operators:

→ Increment Operator is used to increase the value of the Operand by 1.

→ Decrement Operator is used to decrease the value of the Operand by 1.

Operator	Symbol	Example
increment	++	++a (pre) b++ (post)
decrement	--	--c (pre) d-- (post)

Program

```
Class IncDec
{ public static void main(String args[])
  { int a=1, b=1, c=1, d=1;
    S.o.pln(++a); //pre increment //2
    S.o.pln(b++); //post increment //1
    S.o.pln(--c); // pre decrement //0
    S.o.pln(d--); // post decrement //1
  }
}
```

output

```
2
1
0
1
```

* Bitwise Operator:

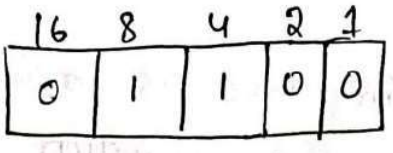
→ A Bitwise operator in Java is a symbol / notation that performs a specified operation on standalone bits, taken one at a time.

→ It is used to manipulate individual bits of a binary number and can be used with a variety of integer types – char, int, long, short, byte.

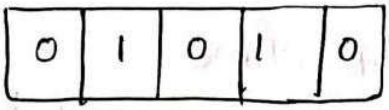
Operator	Symbol	Example a=12, b=10
Bitwise AND	&	a&b = 8
Bitwise OR	|	a|b = 14
← Bitwise XOR	^	a^b = 6
Leftshift	<<	a<<1 = 24
Rightshift	>>	a>>1 = 6

0 0 - 0
1 1 - 0

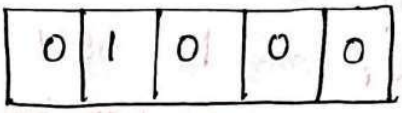
a = 12



b = 10

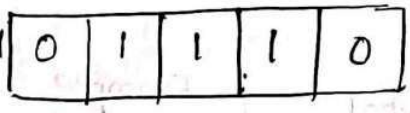


a & b = 8



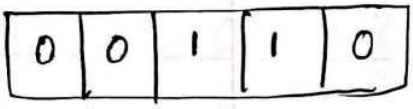
— Bitwise And

a | b = 14



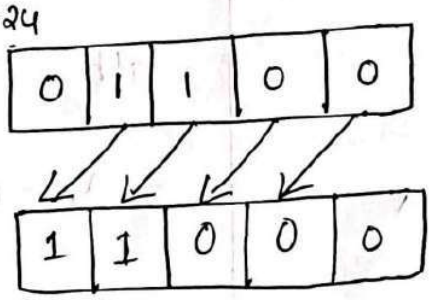
— Bitwise OR

a ^ b = 6



— Bitwise XOR

a << 1 = 24 (n → 1)



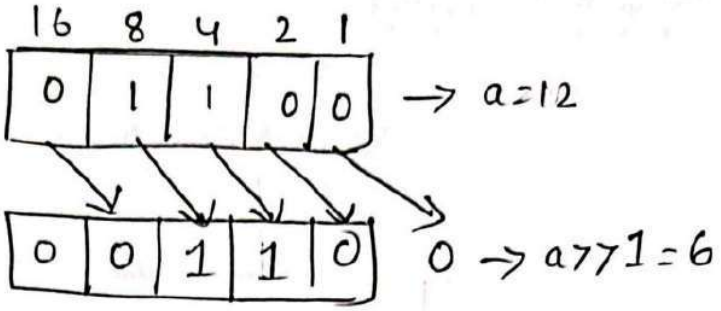
Formula:

a << n = ?

Answer: $2^n * a$ i.e., $2^1 * 12 = 24$

a = 12 n = 1

$a >> 1 = 6$ (n)



Formula:

$a >> n = ?$

Answer:

$a = 12 \quad n = 1$

$$\frac{a}{2^n} = \frac{12}{2^1} = 6$$

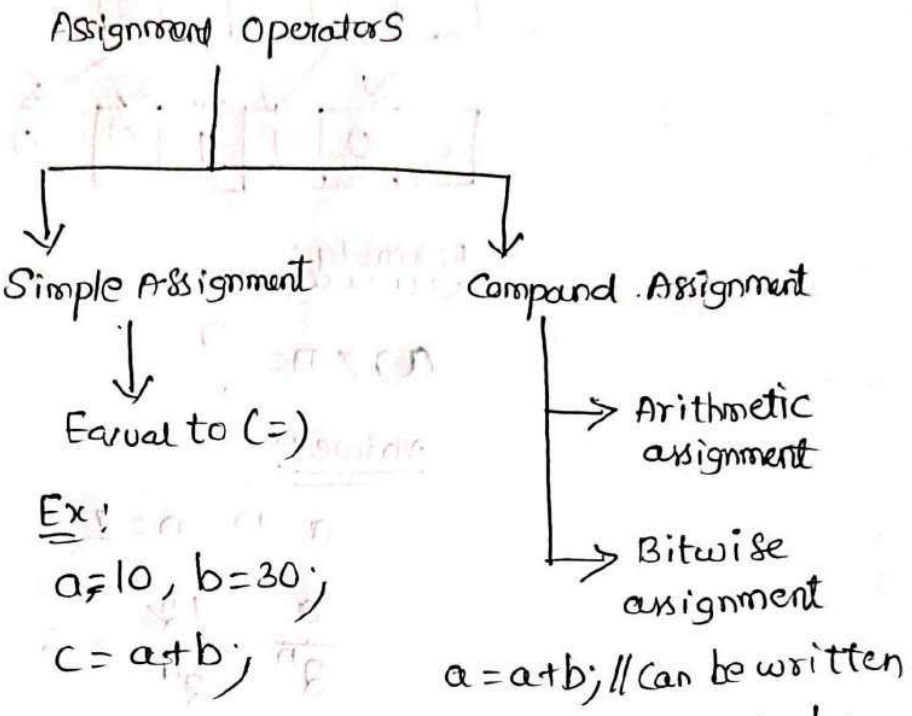
Program

```
Class BitwiseExample
{
  psvm (String args[])
  {
    int a=12, b=10;
    S.O.pln (a&b); → 8
    S.O.pln (a|b); → 14
    S.O.pln (a^b); → 6
    S.O.pln (a<<1); → 24
    S.O.pln (a>>1); → 6
  }
}
```

output

```
8
14
6
24
6
```

* Assignment Operator :. (=)



Ex:

```
a=10, b=30;
c= a+b;
```

```
a = a+b; // Can be written as a+= b;
a = a&b; // Can be written as a&=b;
```

Operator	Symbol	Example
Assignment	=	x=y; and c=a+b;

Arithmetic Assignment

Operator	Symbol	Example
Add AND Equal to	+=	a+=b
Substract AND Equal to	-=	a-=b
Multiply AND Equal to	*=	a*=b
Division AND Equal to	/=	a/=b
Modules AND Equal to	%=	a%=b

Bitwise Assignment

Operator	Symbol	Example
Bitwise AND Equal to	&=	a&=b
Bitwise OR Equal to	|=	a|=b
Bitwise XOR Equal to	^=	a^=b
Bitwise left shift Equal to	<<=	a<<=1
Bitwise Right shift Equal to	>>=	a>>=2

Example

```
class AssignmentExample
{
    psvm (String args[])
    {
        int a=10; //Simple assignment
        S.o.pln ("a="+a);

        int x=5, y=10;
        x+=y; // Compound assignment
        S.o.pln ("x="+x);
    }
}
```

∴ x+=y;
x = x+y;
x = 5+10;
x = 15;

output

a = 10
x = 15

* Separators in Java :-

→ A Separator is a symbol that is used to Separate a group of code from one another.

Parenthesis: → () → used to enuose an argument in the method defination. Also used for defining defination. Also used for defining the expression in control Statement etc.,

Ex :

1) ob.add (5,10);

```
void add (int x, int y)
  {
  }
```

2) if (condition)

```
{
}
```

3) switch (expression)

```
{

}
```

Curly braces :→ { } – used to define a block of code for classes and method. Also used to contain the values of arrays. etc,

Ex: 1) Class Kalam
{
}

2) void Sum(int a, int b)
{
}

3) int arr[] = {1,2,3,4};

Brackets :→ [] – used to declare an array type. Also used when difference array values etc,

Ex: int arr[] = new int[10]; // 1-Dimensional array
(Subscript)

int arr[][] = new int[4][4]; // 2-Dimensional Array

Semicolon: → ; – used to Separate (or) terminate the Statement etc;

Ex:

```
for(k=1; k<=10; k++)

System.out.println("welcome Java");
```

And many more

Comma: → , – used to Separate identifier (or) variable declaration. Etc;

Ex: int x, y, z;

Period or dot: → . – to Separate package names from Sub-packages and Classes. Also used to Separate a variable

Ex: import java.util.Scanner;

obj.add();

NOTE:

→ Semi colon(;), Comma(,), and dot(.). Are also called punctuators

* Literals:

→ Literals in java are a Seavence of characters (digits, letter, and other characters) that represent constant values to be stored in variables.

→ Literals can be any number, text or other information that represents a value. This means what you type is what you get.

→ A literal is the Source code representation of a fixed value.

Example:

```
boolean result = true;
char ch = 'c';
byte b = 200;
Short S = 10000;
int i = 100000;
String name = "vaishnav";
```

datatype

Literal

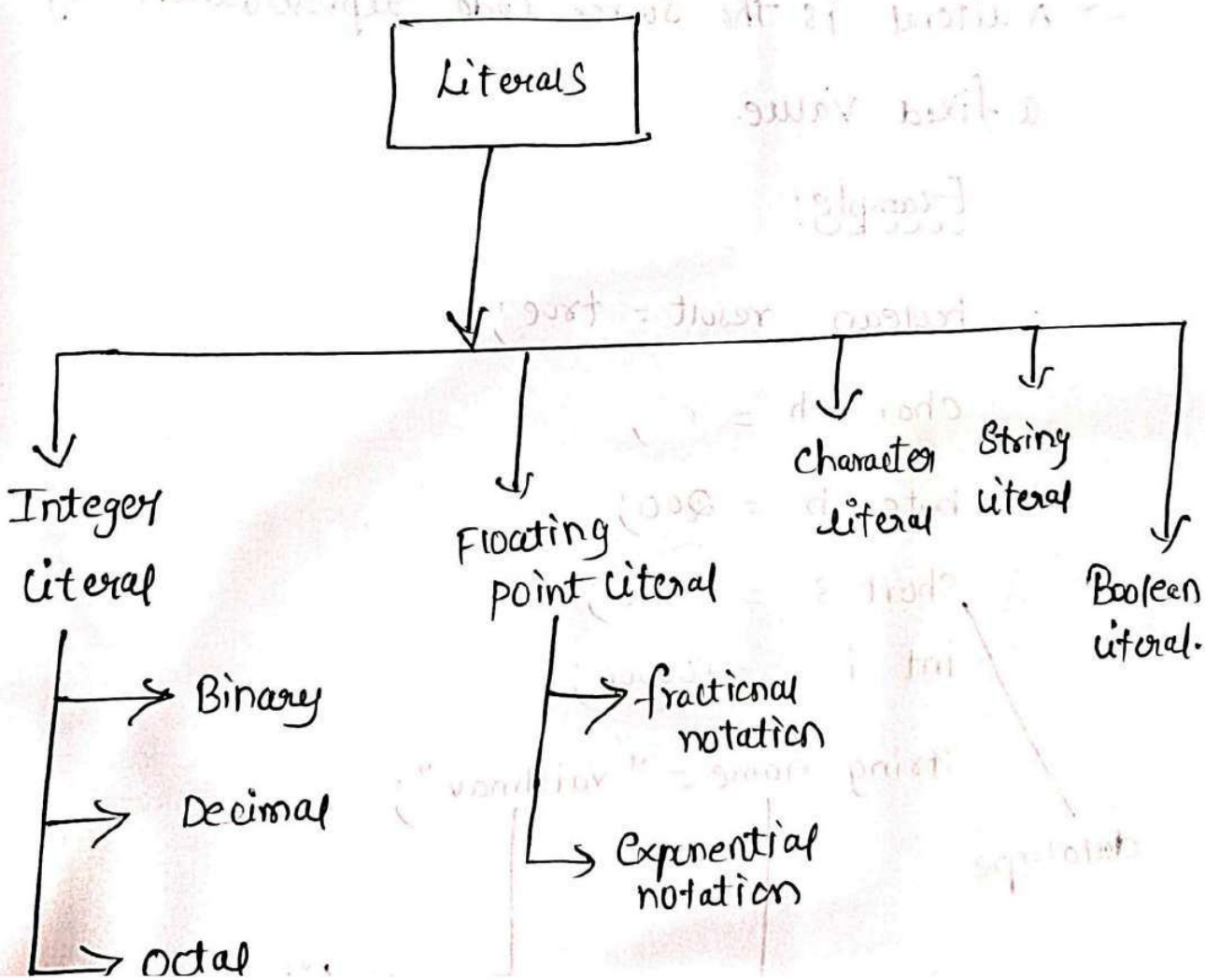
Variable

(or)

→ Literals in java are constant values assigned to variable. Also called constants. It may be number (or) text etc.,

```
int a = 30;
```

- `int` → Datatype
- `a` → name of variable
- `30` → Literal / constant value



Integer literal:

→ It is a numerical value there are 4 types

(1) Binary (Base 2) - ex: int n=10101

(2) Decimal literal (Base 10) - 0 to 9 Ex: 279

(3) octal literal (Base 8) - 0 to 7 Ex: 037

(4) Hexa decimal (Base 16) - 0 to 9 a-f (or) A-F Ex: 0X79AC

Floating point literal:

→ You can represent floating point literal either in decimal form of (or) exponential form.

(or)

→ You can represent floating point literal in two ways.

(1) Fractional notation — 123.45 — float a=1.2f; (It stores --- 4 to 6 digits) F (or)

(2) Exponential notation — double a=1.2d; (It store --- 12 to 14 digits)

$m \times b^{e}$ → e: exponent (not floating point value either +ve integer (or) -ve integer)

m: mantissa (floating point number)

b → 10

Ex: $1.2\,10^{-5}$

$-0.5.7\,e^{-5}$

Character literal:

→ Character literals are unicode character enclosed inside single quotes.

Ex: char ch = 'a'

String literal

→ It is a sequence of characters enclosed inside double quotes

Ex: String name = "Abdul Kalam";

Boolean Literal:

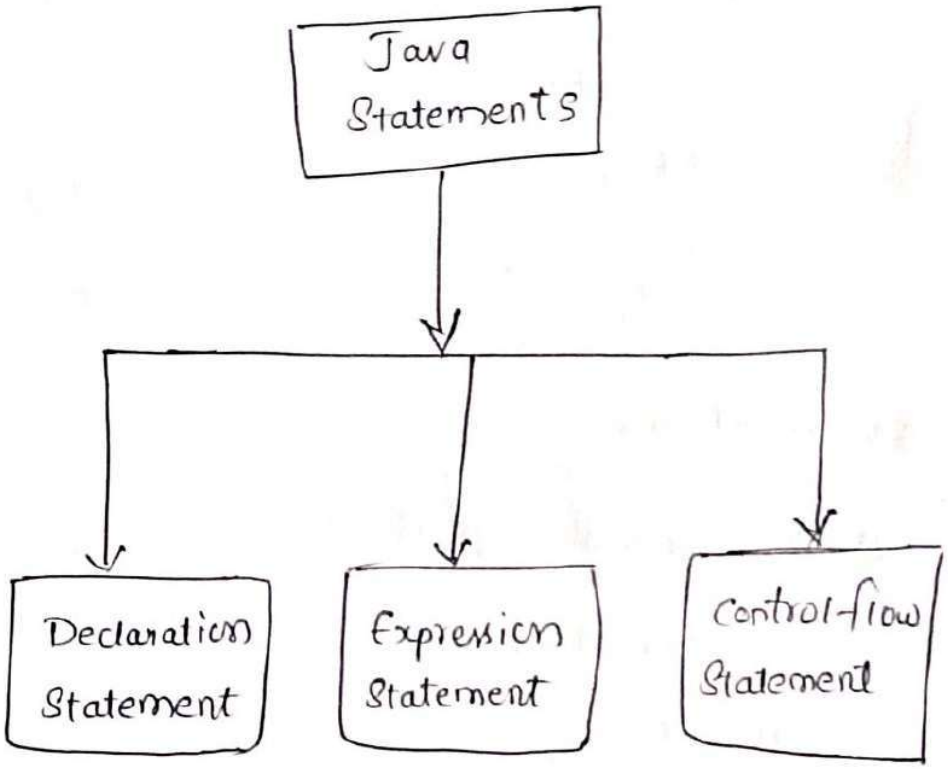
→ It is used to initialize boolean data types i.e., either true (or) false.

Ex: boolean flag = true;

* Java Statements:

→ A Statement Specifies an action in a Java program.

→ Java Statements can be broadly classified into three categories.



(1) Java Declaration Statement:

→ A declaration Statement is used to declare a Variable.

Example:

```
int number;
int num2 = 50;
String str;
```

(2) Java Expression Statement:

→ An Expression with a Semicolon at the end is called an Expression Statement.

Ex:

1) // Increment and decrement Expressions

```
num ++;
++num;
num --;
--num;
```

(3) Assignment Exp

2) // Assignment Expression

```
number = 10;
num *= 10;
```

3) // Method invocation Expressions

```
System.out.println ("This is a Statement");
SomeMethod (param1, param2);
```

Java Flow Control Statement:

→ By default, all Statements in a java program are Executed in the order they appear in the program. Sometimes you may want to execute a set of Statements repeatedly for a number of times (or) as long as a particular condition is true.

→ All of these are possible in Java using flow Control Statements. The if Statement, while loop Statement are examples of control flow Statements.

* Command Line Arguments:

→ The command line arguments in java are the arguments passed to a program at the time when you run it.

→ By default, they are stored in String format.

→ Because we are passing String args[] to main() function.

Program:

```
class Cmdarguments
{
    psvm (String args[])
    {
        System.out.println(args[0]);
        System.out.println(args[1]);
    }
}
```

Save : Cmdarguments.java

Compile: javac Cmdarguments.java

Run : java Cmdargument Hello java

Output Hello
java

/* write a java program addition of two numbers (command line)

```
class cmdarguments
{
  psvm(String args[])
  {
    System.out.println(args[0]+args[1]);//Here "100"+"23" which is String concatenation
  }
}
```

Compile : javac cmdarguments.java

Run : javac cmdarguments 100 50

output: 10050 — Here concatenation

→ By default, they are stored in String format.

* Program:

/* write a java program Addition of two numbers using command line arguments */

→ We need to convert String format to Integer format using parseInt() method

```
import java.lang.*;
Class CmdAddition
{
  Psvm (String args[])
  {
    int a, b;
    a = Integer.parseInt(args[0]);
    b= Integer.parseInt(args[1]);
  System.out.println(a+b);
  }
}
```

Compile: javac CmdAddition.java

RUN: java cmdaddition 100 50

output: 150

* Program:

/* Accepting 'n' number of arguments using command line */

→ We need to use .length in array condition

```
class Cmdarg
{
  psvm (String args[])
  {
    for(int i=0; i<args.length; i++)
    {
      System.out.println(" Java is "+ args[i]);
    }
  }
}
```

Compile: javac cmdarg.java

RUN: java cmdarg good simple Easy

output: java is good
java is simple
java is Easy

* Escape Sequence:

→ A character with a backslash (\) before it is an escape sequence (or) escape character. we use Escape characters to perform some specific task in java.

Escape Sequences in Java:

\n – Inserts a new line

\t - Inserts a horizontal tab

\r – Inserts carriage return

\f - Form feed

\b - Inserts a backe Slash

\' - Single Quote

\" – Double Quoté

\\ – Backe Slash

NOTE:

→ Form feed is a page-breaking ASCII Control of character. It forces the printer to eject the current page and to continue printing at the top of another.

Example

```
import java.lang.*;

class EscapeExample
{
  psvm (String args[])
  {
    S.o.pln ("new \n Line");
      //new
      // Line

S.o.pln ("horizontal\t tab"); //horizontal ___ tab
                                  └space
S.o.pln ("carriage \r return"); // carriage \r return
                                   return
S.o.pln("backspace"); //

S.o.pln(" back\bSpace "); //backSpace

S.o.pln (""Doublequotes""); // Error

S.o.pln ("\"Doublequotes\""); // "Doublequotes"

S.o.pln("\'singleQuote\'"); // 'singlequote'

S.o.pln ("\\backslash "); // \backslash

  }
}
```

* variable and Datatypes in Java:

Variable:

→ A variable is a container which holds the value while the java program is executed.

(or)

→ variables are Data containers that Save the data values during java program execution.

→ Every variable in java is assigned a datatype that designates the type and quanity of value it can hold.

→ A variable is a memory location name for the data.

Declaring variable:

→ To create a variable, you must Specify the type and assign it a value.

Syntax:

```
type variable = value;
```

where type is one of Java's types (Such as int, float, String etc) and variable is the name of the variable (such as Identifier).

*[redacted]

In Java, there are different types of variables, for example.

→ String – Stores text, such as "Java". String value surrounded by double Quotes.

→ int – Stores integers (whole numbers), without decimal, such as 132 (or) -132

→ float – Stores floating point numbers, with decimal, such as 23.72 (or) -23.72

→ char – Stores single characters, such as 'a' (or) 'D'. Char values are surrounded by single Quotes.

→ boolean – Stores values with two States: true (or) false

Ex:

```
int num1 = 10;
float num2 = 6.89f;
Char ch = 'k';
boolean status = true;
String name = "vaibhav";
```

```
System.out.println("value inside Integer type variable "+num1);
System.out.println("value inside float type variable "+num2);
System.out.println("value inside character type variable"+ch);
System.out.println("value inside boolean type variable "+b);
System.out.println("value inside String type variable"+name);
```

Declare Many Variables:

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

Ex `int x=1, y=2, z=3;`

```
System.out.println(x+y+z);
```

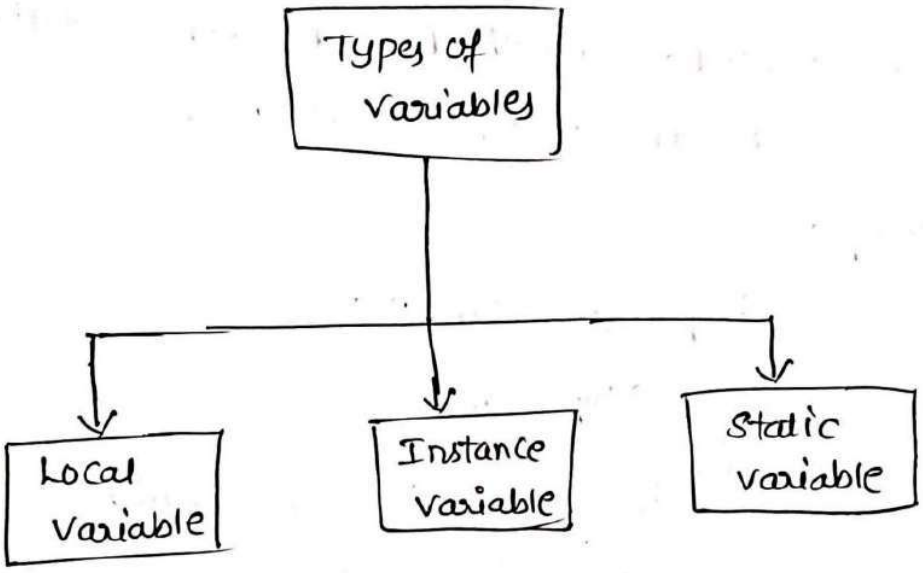
Program:

```
// write a java program different types of variable
class variables
{
    psvm(String args[])
    {
        int num1=9;
        float num2 = 7.6f;
        char ch = 'd';
        boolean status = true;
        String name = "Kalam";
    S.o.pln ("value inside integer type variable:"+num1);
    S.o.pln ("value inside float type variable:"+num2);
    S.o.pln ("value inside character type variable:"+ch);
    S.o.pln ("value inside boolean type variable:"+status);
    S.o.pln ("value inside String type variable:"+ name);
    }
}
```

Output:

value inside integer type variable : 9
value inside float type variable : 7.6
value inside character type variable : d
value inside boolean type variable : true
value inside String type variable : Kalam

*Types of Variables:



Local Variable:

→ A variable which is declared inside a class and inside a method is called as "Local Variable."

Example

```
import java.lang.*;
Class Myprogram
{ psvm(String args[])
  {
    int a=50;//Local Variable
    S.o.pln(a);
  }
}
```

output:

50

Instance Variable

→ A variable which is declared inside a class and outside a method and without static keyword is called as Instance Variable.

Example

```
import java.lang.*;
Class Myprogram
{
    int x=40; // Instance Variable
    psvm (String args[])
    {
    Myprogram ob = new Myprogram();
    System.out.println(ob.x);
    }
}
```

output:

40

Static Variable:

→ A variable which is declared inside a class and outside a method and with "Static keyword."

Example:

```
import java.lang.*;

class Myprogram
{
    static int a=100;
    psvm(String args[])
    {
    System.out.println(a); (or)
    }                          System.out.println(
}                                  myprogram.x);
```

Output:

100

* Datatype :

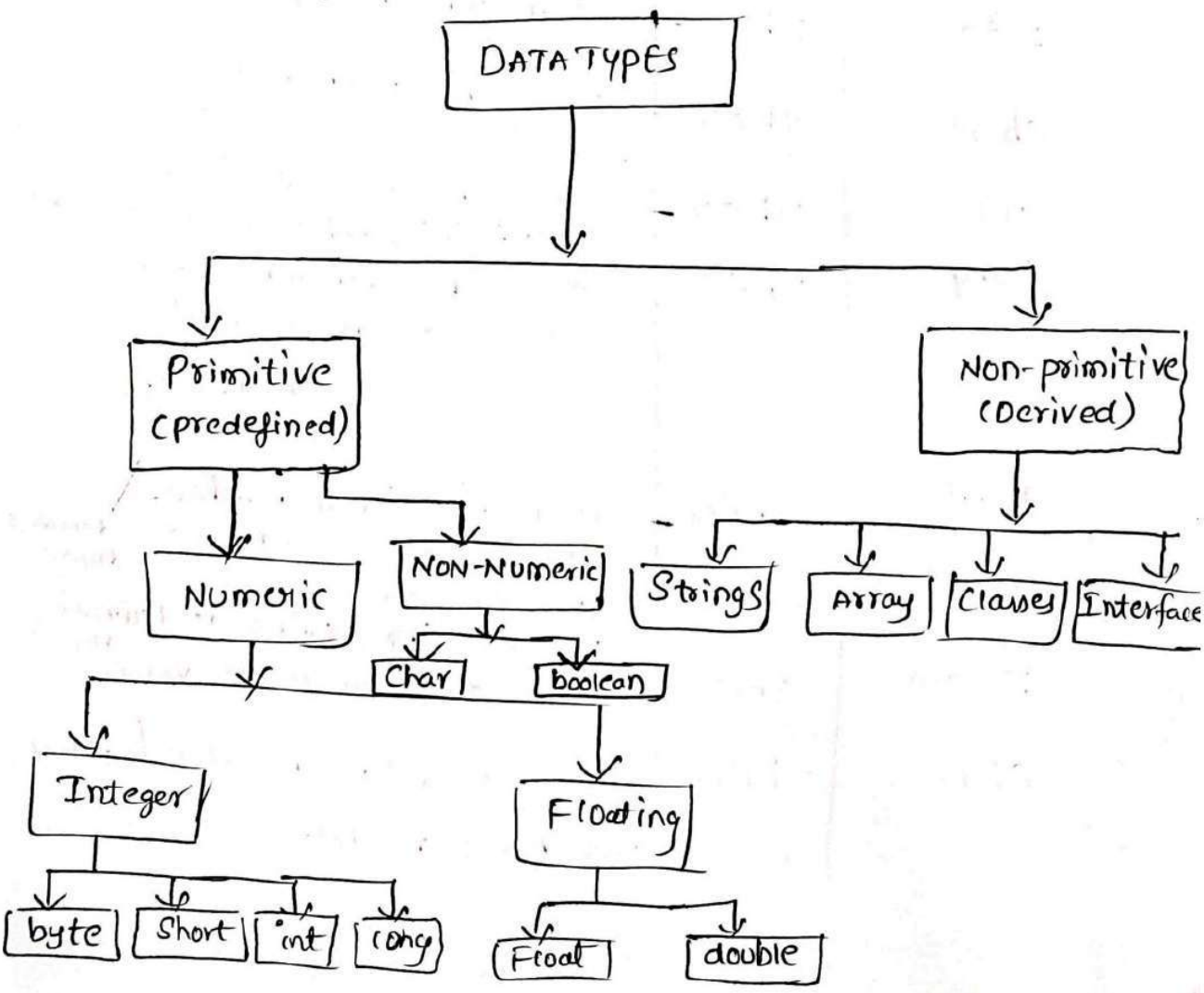
→ Datatype defines what type of data Stored into a variable.

→ It is a representation of data.

→ Datatype associated with two things.

(i) what type of data allowed to Store

(ii) How much memory is allocated to that data.



Primitive Data Types:

→ A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive datatypes in java.

Datatype	Size	Description
Byte	1 byte	Store whole numbers from -128 to 127
Short	2 bytes	Store whole numbers from -32,768 to 32,767
Int	4 bytes	Store whole numbers from -2147,483,648 to 2,147,483,647
Long	8 bytes	Store whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,807
Float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits.
boolean	1 bit	Store true (or) false values
Char	2 bytes	Stores a single character / letter for ASCII values.

Non-primitive Data type :

→ Non-primitive Data types are called reference types because they refer to object.

→ primitive types are defined (already defined) in java. Non primitive types are created by the programmer and is not defined by java (except for String)

→ Examples of non-primitive types are Classes, Interface, Strings, Arrays etc.

String:

→ The String datatype is used to store a sequence of characters (text). String values must be surranded by double Quotes.

Ex: String name = "Kalam";

System.out.println(name);

Note :-

→ A String in java is actually a non-primitive datatype, because it refers to an object.

→ The String object has methods that are used to perform certain operations on Strings.

Program:

// write a java program different types of variable (or Datatypes)

```
class variables
{
    psvm(String args[])
    {
        int num1=9; // Integer Datatype
        float num2 = 7.6f; // float DT
        char ch = 'd'; // char DT
        boolean Status = true; // boolean DT
        String name = "Kalam"; // String DT

        S.o.pln ("value inside integer type variable:"+num1);
        S.o.pln ("value inside float type variable:"+num2);
        S.o.pln ("value inside character type variable:"+ch);
        S.o.pln ("value inside boolean type variable:"+Status);
        S.o.pln ("value inside String type variable:"+ name);
    }
}
```

Output:

value inside integer type variable : 9
value inside float type variable : 7.6
value inside character type variable : d
value inside boolean type variable : true
value inside String type variable : Kalam

* Static variable & Static methods:

→ The Static keyword in java is used for memory management. It makes your program memory efficient [i.e it saves memory].

→ It can apply for variables, methods, blocks.

→ The Static variable can be used to refer the common property of all objects.

→ Static variables gets memory only once in class area at the time of loading.

* Program: // Display Student Details

```
import java.lang.*;
class Student
{
    int rollno; // Instance Variable
    String name; // Instance Variable
    Static String college = "Aditya"; // Static Variable

    Student(int r, String n) // Constructor
    {
        rollno = r;
        name = n;
    }
    void display() // Instance Method
    {
        S.o.pln(rollno + " " + name + " " + college);
    }
    public Static void main (String[] args)
    {
        Student St1 = new Student(500, "vaibhav");
        Student St2 = new Student(501, "vaishnav");
        St1.display();
        St2.display();
    }
}
```

Output:

```
500  vaibhav  Aditya
501  vaishnav  Aditya
```

* program : // Display Student Details

```
import java.lang.*;

Class Student
{
    int rollno; // Instance variable
    String name; // Instance variable
    Static String College = "Aditya"; // Static variable
    Student( int r, String n) // Constructor
    {
        rollno = r;
        name = n;
    }
    Static void change() // Static method
    {
        College = " Sai Aditya";
    }
    void display() // Instance method
    {
        S.o.pln(" rollno + " " + name + " " + college;
    }
    public Static void main (String args[])
    {
        Student.change();
        Student St1 = new Student (500, "vaishnav");
        Student St2 = new Student (501, "vaibhav");
        St1.display();
        St2.display();
    }
}
```

Output :

500 vaishnav SaiAditya
501 vaibhav SaiAditya

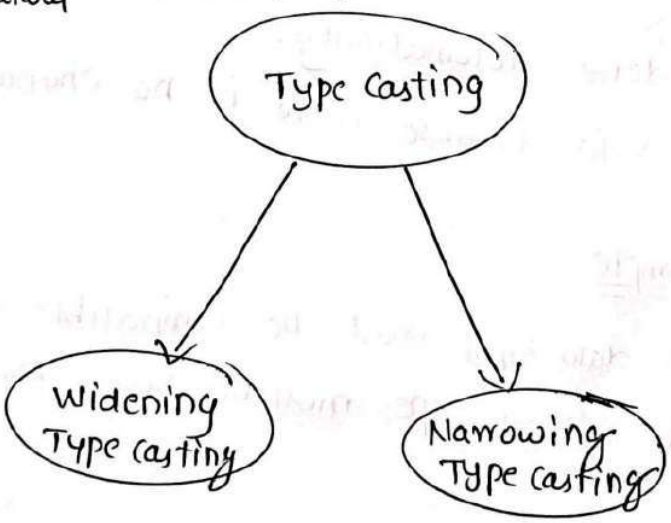
* Type Casting:

-> It is used to convert one datatype to another, boolean data type not supports it.

(or)

-> In Java, type casting is a method or process that converts a data type into another datatype in both ways manually and automatically.

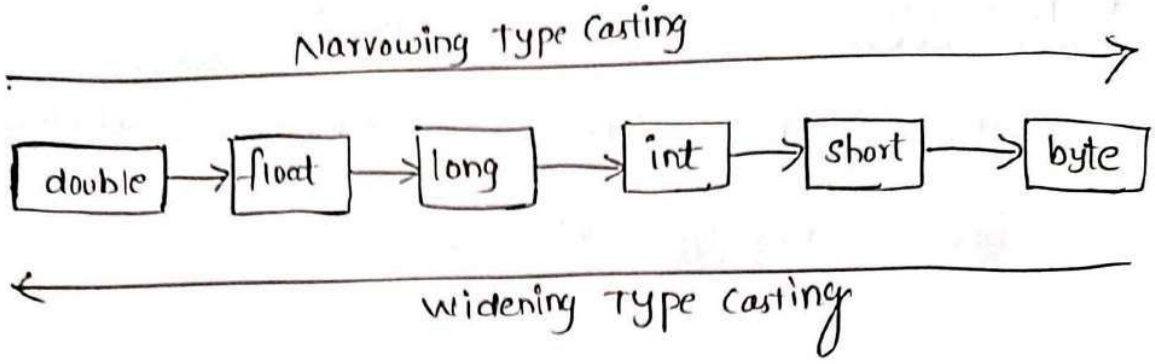
-> The automatic conversion is done by the compiler and manual conversion performed by the programmer.



There are two types of type casting.

(i) widening Type Casting

(ii) Narrowing Type Casting



Type Casting in java

widening Type Casting:

→ Converting a lower data type into a higher one is called widening type casting. It is also known as implicit conversion (or) casting down.

→ It is done automatically.

→ It is safe because there is no chance to lose data.

Example

→ Both data types must be compatible each other.

→ The target type must be larger than source type.

byte → short → char → int → long → float → double

* Example: WideningTypeCasting.java

```
import java.lang.*;

Class WideningTypeCastingExample
{  psvm (String args[])
   {
      int x=9;
   //automatically converts the integer type into long type
      long y = x;
//automatically convert the long type into float type
      float z = y;
      S.o.pln("Before conversion int value"+x);
      S.o.pln("After conversion, long value"+y);
      S.o.pln("After conversion, float value"+z);
   }
}
```

Output:

Before Conversion, the value is : 9

After Conversion, the long value is : 9

After conversion, the float value is : 9.0

Note: In the above Example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

(ii) Narrowing Type Casting:

→ Converting a higher datatype into a lower one is called narrowing type casting. It is known as explicit conversion (or) casting up.

→ It is done manually by the programmer.

Example: NarrowTypeCasting Example.java

```
import java.lang.*;

Class NarrowingTypeCastingExample
{ psvm (String args[])
  {
    double d = 197.88
  //converting double datatype into long datatype
    long l = (long) d;
  //converting long datatype into int datatype
    int i =(int)l;
  S.o.pln("Before conversion:"+d);
    // fractional part lost
  S.o.pln (" After conversion into long type:"+l)
    //fractional part lost
  S.o.pln ("After conversion into int type:"+i);
  } }
```

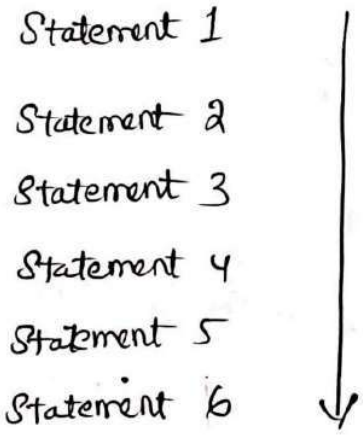
Output

```
Before conversion: 197.88
After conversion into long type: 197
After conversion into int type: 197
```

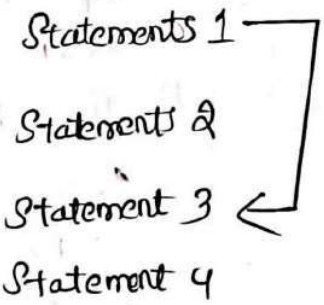
* Control Statements : (or) Control flow Statements

→ Control flow Statements are used to control the program's flow of execution also called control structure.

→ Generally programs are executing Statements in Seqvence.



→ Sometimes to stop normal flow and jump to another Statements.



→ To execute single statement (or) block repeatedly.

Statement 1

Statement 2

Statement 3

Statement 4

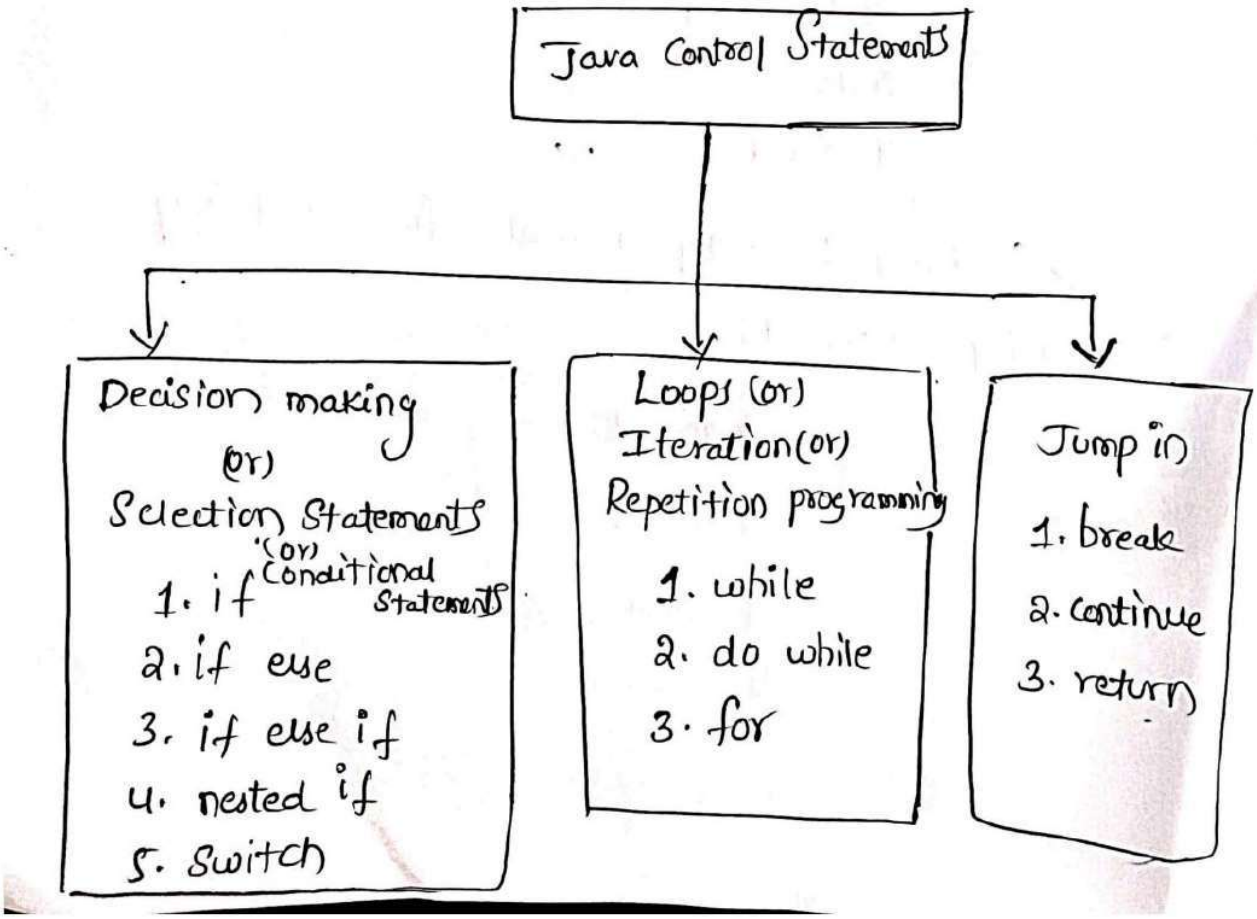
→ to break and skip repetition

Statement 1

Statement 2

Statement 3

Statement 4



Some situations where programmer have to change the order of execution of statements based on certain conditions which involve kind of decision making statement.

List of decision making statements in java.

(i) if (Simple if)
(ii) if else
(iii) if else if (or if else ladder)
(iv) nested if
(v) Switch

Note:

→ if else statement is a two way branch statement. Depending upon the whether a condition is true (or) false, the corresponding code is executed.

→ else if ladder, Switch statement is a multiway branch statements. Depending upon condition (or) expression used the corresponding code is executed.

(i) if (or) Simple if:

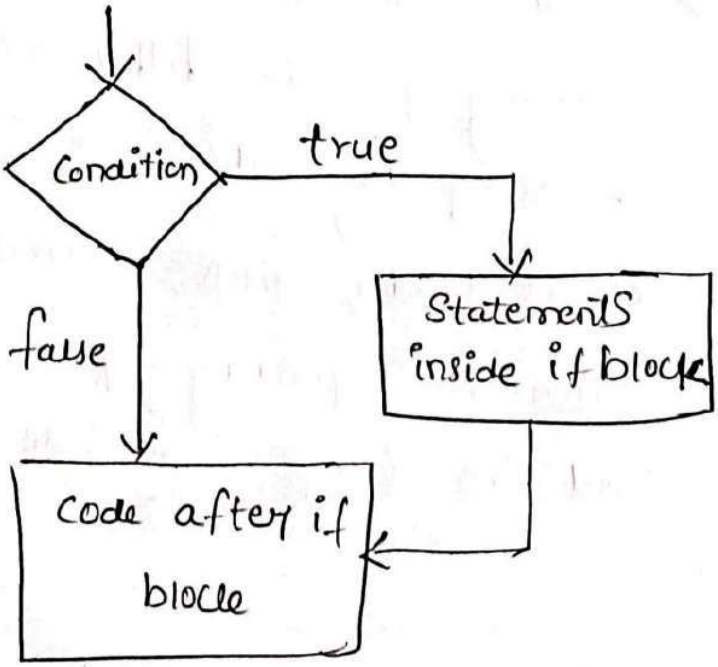
→ if test condition is true then block of statements will be executed.

i.e., it tells compiler to execute certain part of code only if particular test condition is true.

Syntax:

```
if (test condition)
{
  // statements
}
```

Flow chart:



Example : 1

```
import java.util.*;

Class IfExample
  {
    public static void main (String args[])
      {
Scanner s = new Scanner(System.in); // to read
                                    value at command prompt
   int number;
S.opln (" Enter a number ");
number = SC.nextInt(); // ready value and
                          store in variable
                          number
        if (number > 0)
          {
            s.o.pln (" positive number ");
          }
      }
```

Output:

```
Enter a number :
100
 positive number
```

Example 2:

```
import java.util.*;

Class IfExample
{
    psvm(String[] args)
    {
        int number = 90; //local variable

        if(num > 0) // condition
        {
            S.o.pln("positive number");
        }
    }
}
```

output:

positive number

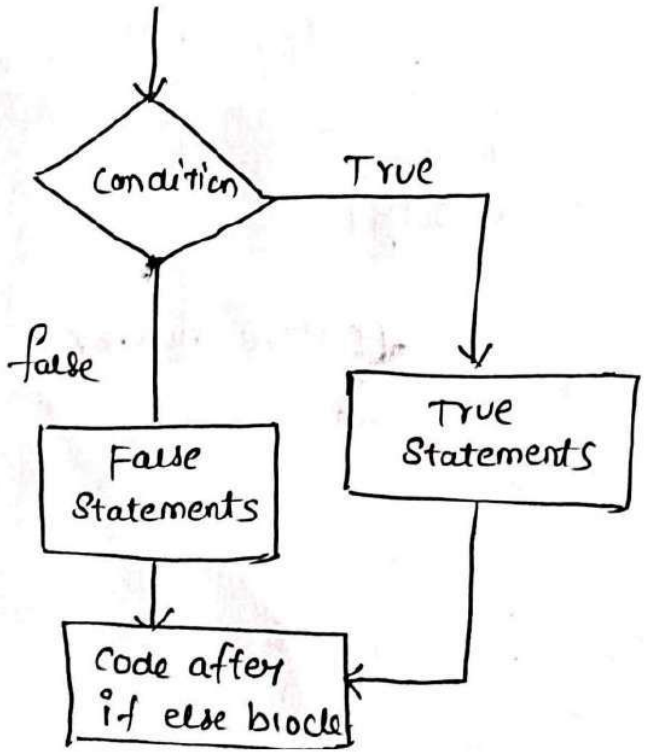
(ii) if else :

→ Here when test condition is true then true block statements will be executed otherwise false block statements will be executed.

Syntax :-

```
if (test condition)
{
  //true Statements
}
else
{
  //false Statements
}
```

Flow chart :



Example

```
import java.lang.*;

Class IfElseExample
{
   psvm(String args[])
    {    int num = 7;
      if (num >7) // 7 > 0 → true
       {
       S.o.pln(" positive number ");
        }
        else
        {
          S.o.pln(" Negative number ");
          }
    }
}
```

Output:

Positive number

*// Java program to check biggest among 2 numbers

```
import java.lang.*;

class Biggest
{
    psvm(String args[])
    {
        int x=10, y=5;
        if (x>y) // 10>5 → True
        {
            S.o.pln(x + " is big");
        }
        else
        {
            S.o.pln(y + " is big");
        }
    }
}
```

output:

10 is big

Example:

// write a java program to check given character is vowel (or) constant using if else

```
import java.lang.*;

Class VowelConstant
{
   psvm(String args[])
   {
      char ch = 'a';
      if (ch=='a'||ch=='e'||ch=='i'||
          ch=='o'||ch=='u'||ch=='A'||
          ch=='E'||ch=='I'||ch=='O'||ch=='U')
      {
         S.o.pln ("vowel");
      }
      else
      {
         S.o.pln ("consonant");
      }
   }
}
```

output

vowel

(iii) if else if (or) else if ladder

→ It is extension for if else Statement, Here it will check one condition after another in a Sequence.

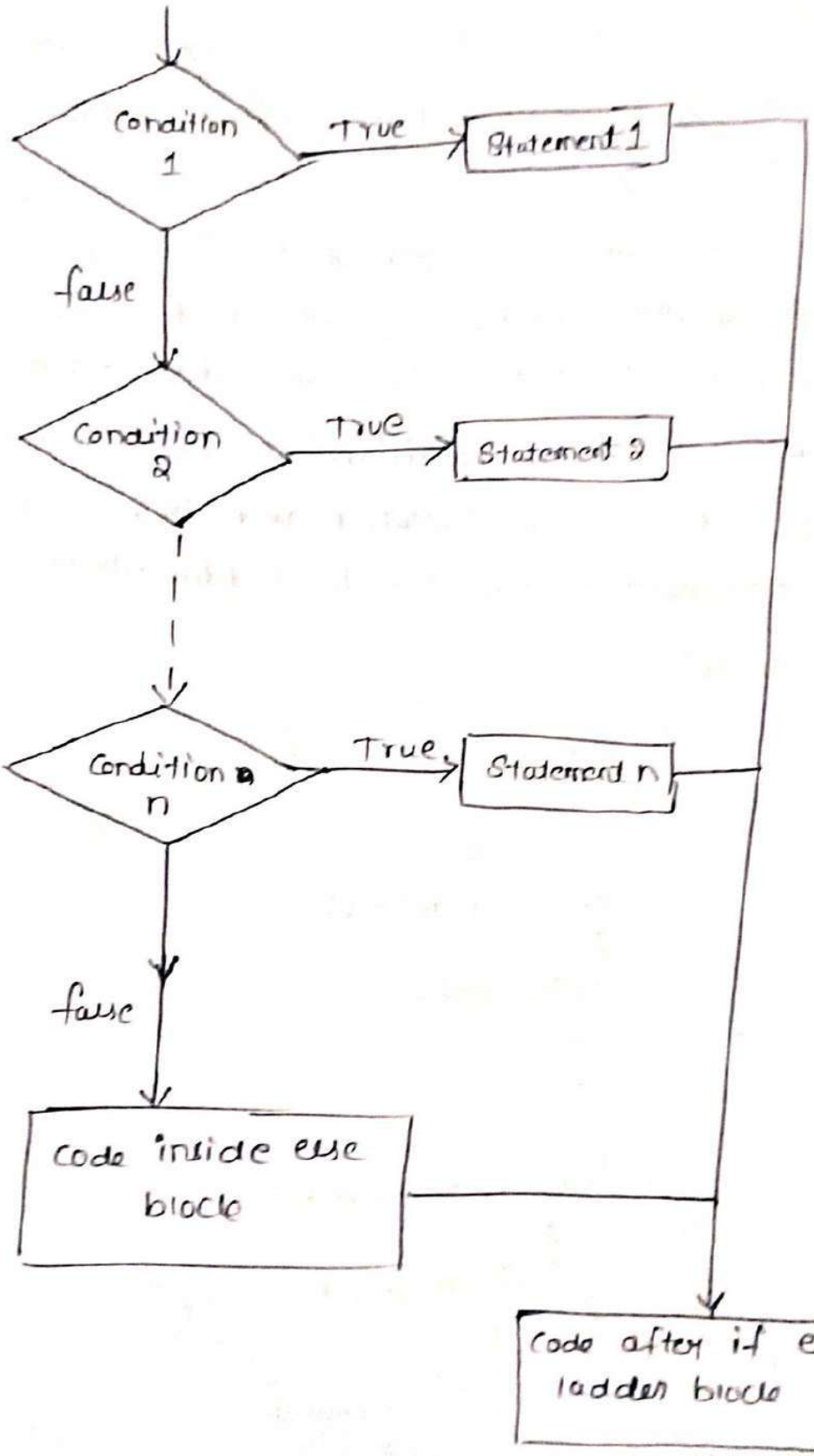
→ In that way if condition is true then corresponding Statements will be executed and control goes to Statements after else if ladder block

→ At last if no condition is true then else block ~~placed~~ will be executed and control goes to Statements after else if ladder block.

Syntax:

```
if (condition 1)
  {
    //Statements 1
  } else if (condition 2)
    {
      //Statement 2
    }
    :
    :
    else if (condition n)
      {
        //Statements n
      }
      else
        { //Statements
        }
```

Flow chart:



Example:

```
import java.lang.*;
class ElseLadderExample
{
   psvm(String args[])
     {
        int n=5;
       if(n==0) //false
       { S.o.pln("ZERO");
       }
       else if(n==1) //false
       {  S.o.pln("one");
          }
       else if(n==2) //false
       { S.o.pln("Two");
       }
       else if(n==3) //false
        {  S.o.pln("Three");
        }
        else if(n==4) //false
         {  S.o.pln("Four");
          }
          else if(n==5) //true
           {  S.o.pln("Five");
            }
```

```
else if (n==8)
{
    S.o.pln("Eight");
}
else if(n==9)
    {S.o.pln("Nine");
     }
     else
     {
      S.o.pln("Not a Single Digit");
     }
   }
}
```

Output

Five

(iv) Nested if:

→ If one if condition is placed inside another if condition we call it as nested if condition.

→ The contained if Statement is known as inner if Statement and another is known as outer if Statement. The inner block of if Statement will be executed only if the outer block condition is true.

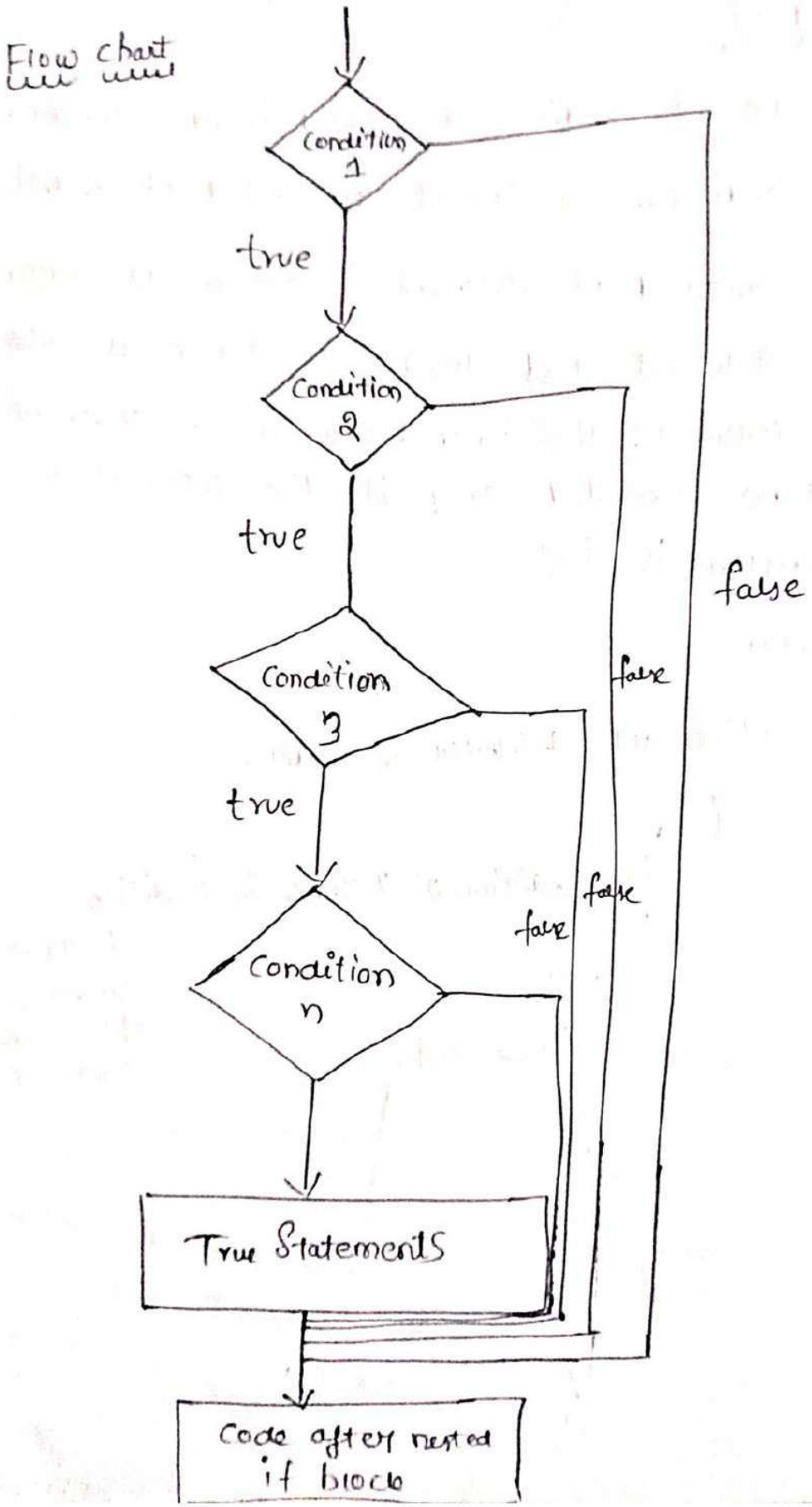
Syntax:

```
if (condition1) //outer if condition
{
    if (condition 2) // Inner if condition
    {
      //true Statements
    }
}
```

If outer condition is True then control goes to check inner condition

If all conditions are true then Statements will be executed

Flow chart


Condition 1
true
Condition 2
true
Condition 3
true
Condition n
false
false
false
false
True Statements
Code after nested if block

(v) Switch case:

→ Switch case control Statement is used to execute specific block of statements, in given number of blocks.

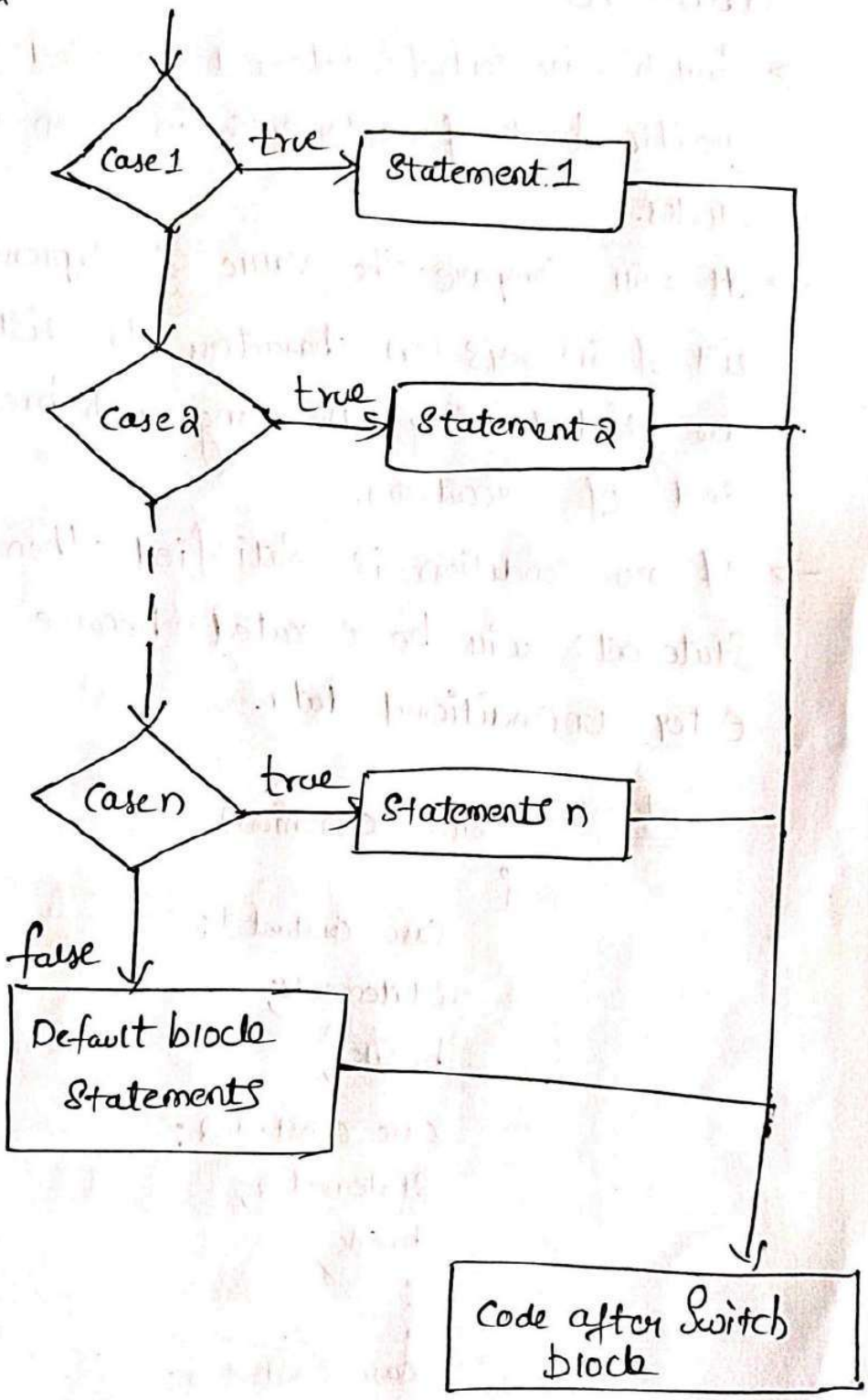
→ It will compare the value of expression against list of integers (or) characters the list of constants are listed using case along with break at the end of execution.

→ If no condition is satisfied then default statements will be executed because user may enter unconditional data.

Syntax:

```
Switch (expression)
{
    case constant 1:
    Statements;
    break;
    case constant 2:
    Statement 2;
    break;
    :
    :
    case Constant n:
    Statements n;
    break;
    default:
    default Statements;
    break; }
```

Flow chart



* Example:

```
import java.lang.*;
Class SwitchExample
{
  psvm(String args[])
  {
     int n=5;
     Switch(n)
     { case 0:
         S.o.pln("ZERO");
       break;
  case 1:
  S.o.pln("ONE");
  break;
  case 2:
  S.o.pln("TWO");
  break;
case 3:
  S.o.pln("THREE");
  break;
  case 4:
  S.o.pln("FOUR");
  break;
  case 5:
  S.o.pln("FIVE");
  break;
  case 6: ~~("SIX~~
  S.o.pln("SIx");
  break;
  case 7:
   S.o.pln("SEVEN");
   break;
```

```
        case 8:
          S.O.pln ("EIGHT");
          break;
        case 9:
          S.O.pln ("NINE");
          break;
          default:
        System.out.println ("NOT A SINGLE DIGIT");
          break;
          }
      }
    }
```

Output:

FIVE

Example:

* //write a Java program vowel (or) constant Switch Case

```
import java.lang.*;

class VowelConstantExample
{
    psvm(String args[])
    {
        char ch = 'a';

        Switch (ch)
        {
          case 'a': case 'A':
          case 'e': case 'E':
          case 'i': case 'I':
          case 'o': case 'O':
          case 'u': case 'U':
          System.out.println("Vowel");
          default:
          System.out.println("consonant");
          break;
        }
    }
```

Output:

Vowel

Example:

// write a Java program String case with String literal

→ which is not there in C and C++

→ String must be not NULL

```
import java.lang.*;

class StringSwitchDemo
{ psvm (String args[])
  {
    String st = "Java";
    Switch (st)
    {
    case "Java":
    S.o.pln ("Java Text Book);
    break;
    case "python":
    S.o.pln (" python Text Book);
    case "Devops":
    S.o.pln ("Devops Text Book");
    default:
    S.o.pln ("wirong choice");
    breale; }
```

Output

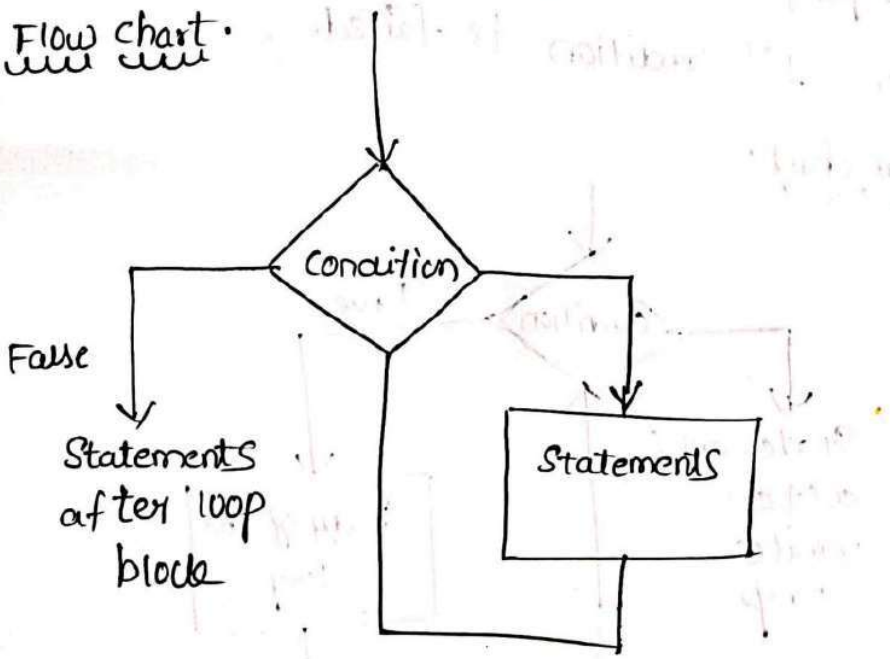
Java

* Loops (or) Iteration (or) Repetition Programming:

In Java, when you want to execute a specific statement (or) statements block number of times until given condition is true we need to use the following.

(i) while
(ii) do while (or) do loop
(iii) for

Flow chart:



* while :

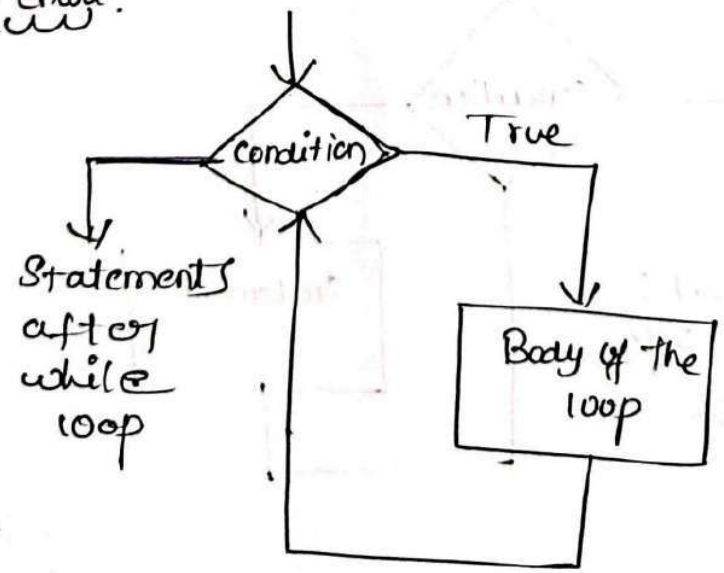
→ while loop is used when we don't know number of repetitions before starting body of the loop.

Syntax :

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

→ Looping will not be continued even at once when 1st condition is failed.

Flow chart :

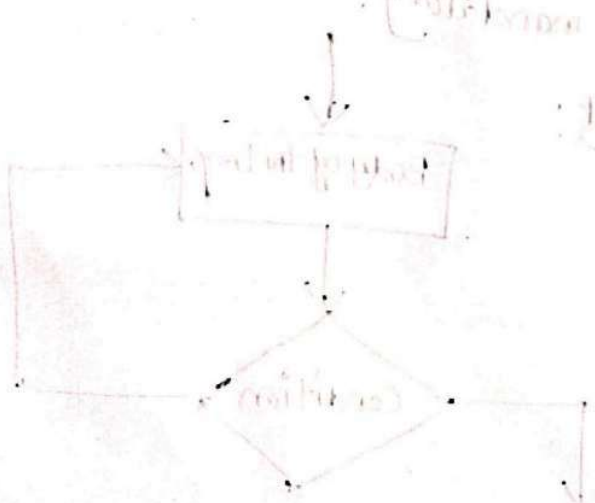


* Example:

```
class whileExample
{
  psvm(String args[])
  { int i=1;
   while(i<=100)
   {
    S.o.pln(i);
    i++;
   }
  }
}
```

Output:

1
2
3
:
:
100



* do while (or) do loop:

→ Do while loop in java programming is used to execute body of the loop atleast once irrespective of given condition.

→ Here it will execute body once and then checks the condition.

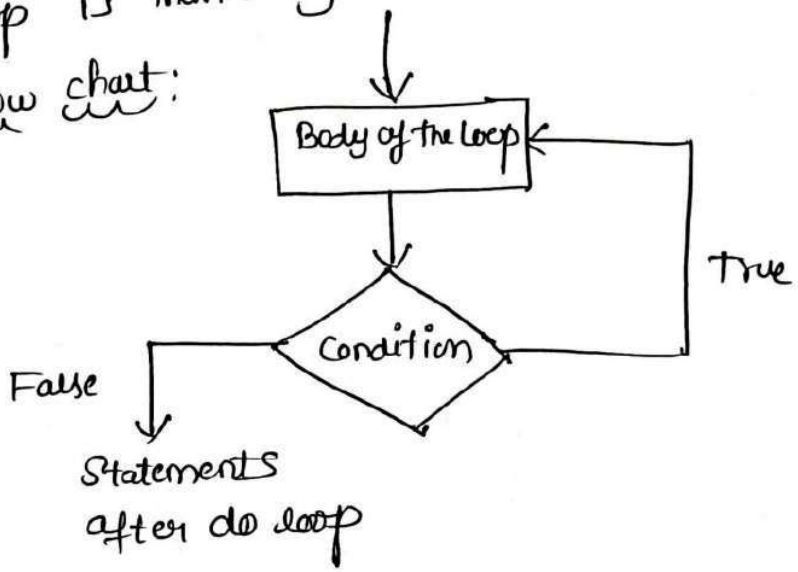
Syntax:

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

Note:

→ Semi colon(;) after condition is do while loop is mandatory.

Flow chart:



* Example

→
```
class DowhileExample
{ psvm(String args[])
  {
    int i=1;
    while(i7=100) // false
    {
      S.o.pln(i);
      i++;
    }
  }
}
```

output:

?

No output - Starting while condition is false.

→
```
class DowhiteExample
{
  psvm(String args[])
  { int i=1;
    do
    { S.o.pln(i);
      i++;
    } while(i7=100);
  }
}
```

output:

1

→ //print 1 to 100 numbers (do while)

```
class DowhileExample
{psvm (String args[])
{
  int i=1;
  do
  {
    S.o.pln (i);
    i++;
  } while (i<=100);
}
}
```

O/P :

```
1
2
3
4
:
:
100
```

* for loop:

→ It is used to execute when we know number of repetitions before starting body of the loop.

Syntax:

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

Ex:

```
for (int i=1; i<=50; i++)
```

Initialization:

→ It holds starting value of loop ex: int i=1

Condition:

→ checks up to which point looping will be continued ex: i<=50

Updation:

→ updates value of intialized value either increment (++) or decrement (--). Ex: i++, i--

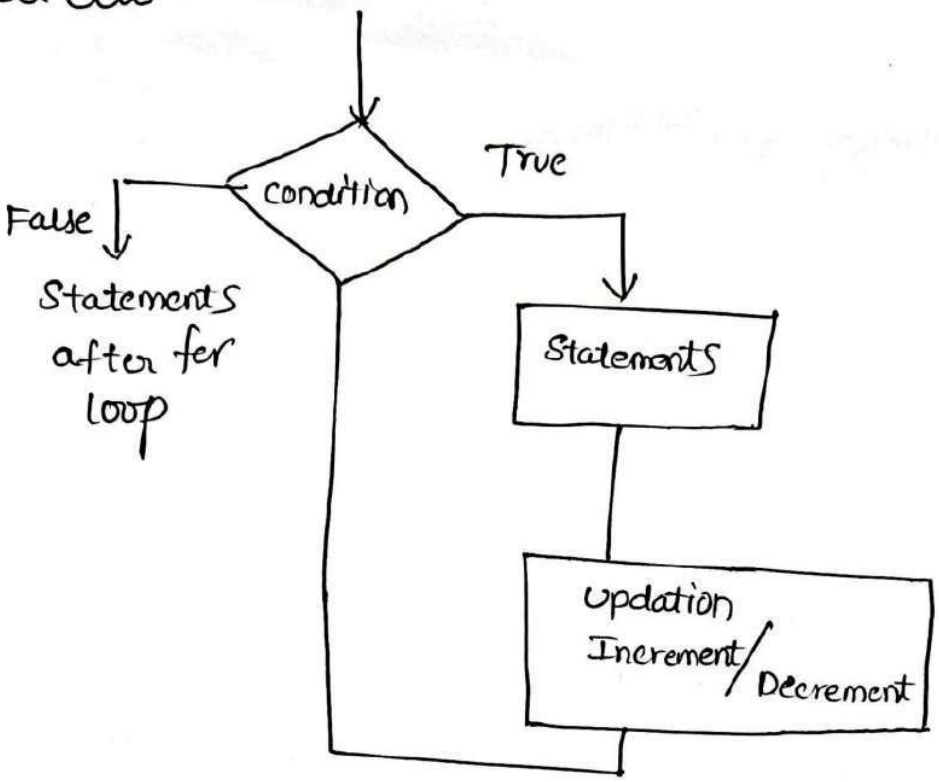
// write a java program to print 1 -50

```
class ForExample
{
    psvm(String args[])
    {
        for(int i=1; i<=50; i++)
        {
            s.o.pln(i);
        }
    }
}
```

output

```
1
2
3
4
5
.
.
.
49
50
```

Flow chart:



// program to print 100 to 1

```
class ForExample
{ psvm (String args[])
  {
    for(int i=100; i>=1; i--)
    {
      S.o.pln(i);
    }
  }
}
```

output 100
99
98
:
:

* Nested for :

→ If one for loop is placed inside another for loop is called nested for loop.

→ It contain one outer for loop and one or more Inner for loops.

Example :

```
for(int i=1;i<=5;i++)
  {
    for(int j=1;j<=5;j++)
      {
        System.out.print(i+""+j+" ");
      }
    System.out.println();
  }
```

output :

```
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
41 42 43 44 45
51 52 53 54 55
```

Example:

```
import java.lang.*;

Class NestedForExample
{
    psvm(String args[])
    {
        for(int i=1 ; i<=5; i++)
        {
            for(int j=1 ; j<=5 ; j++)
            {
                S.o.p(i+" "+ j+" ");
            }
            S.o.pln( );
        }
    }
```

Output:

```
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
41 42 43 44 45
51 52 53 54 55
```

* for each loop:

Syntax:

```
for ( TYPE VAR-NAME : ArrayList / Array)
        { ... }
```

```
int [] arr = {4, 8, 9, 1, 10, 5, 3, 2};
```

```
for (int i=0; i<= arr.length-1; i++)
  {
    int y = arr[i];
    s.o.p(y);
  }
```

```
for(int y : arr)
  {
    s.o.p(y);
  }
```

→ Simple Syntax they have given for that is foreach loop for just colon from which array you want to process the element arr.

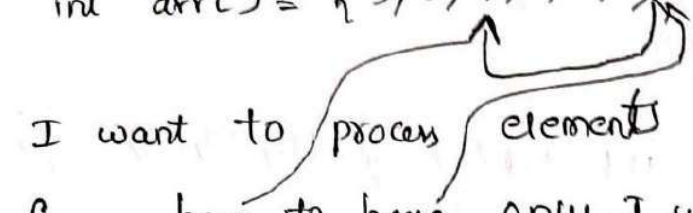
→ No need to specify the starting index, ending index and modification increment (or) decrement.

→ From this array automatically element will come and store into one variable what we have given suppose 'y'.

Limitations

→ If you want to process elements in reverse order it is impossible. you must go with for loop only. we cannot use for each loop.

→ `int arr[] = {3, 8, 7, 9, 2, 6, 3, 4};`



I want to process elements from Specific Set from here to here only, I want to process no in that case also for Each loop is not possible.

* Example:

```
// Example of Java for each loop
class ForEachExample
{
    psvm(String args[])
    {
        int arr[] = {10,20,30,40};
        for(int i : arr)
        {
            S.o.pln(i);
        }
    }
}
```

Output

```
10
20
30
40
```

* For Each loop Example:

```
// Traversing the collection elements

import java.util.*;

Class ForEachEx
{
    psvm(String args[])
    {
    // creating a list of elements

ArrayList<String> list = new ArrayList<String>();

    list.add("apple");
    list.add("orange");
    list.add("grape");

//traversing the list of elements using
//for each loop

    for(String str : list)
    {
        System.out.println(str);
    }
}
}
```

output:

apple
orange
grape

*Symbolic constant in java:

→ In Java, a Symbolic constant is a named constant value defined once and used throuout a program. Symbolic constants are declared using the final keyword.

↳ which indicates that the value cannot be changed once it is initialized.

↳ The naming convention for Symbolic constants is to use all capital letter with underscores separating words.

Syntax of Symbolic constants

```
final <data-type> <variable_name> = value;
```

Ex:

```
final double PI = 3.14;
final int MAX_MARKS = 400;
```

→ final variables are declared by using final keyword.

→ final variables should be initialized in a class at the time of declaration (or) inside the constructor.

Ex 1:

```
Class FinalVarExample
{
    final int MAX_MARKS; // variables should be initialized
}
```

Error

Ex 2:

```
Class FinalVarExample
{
    final int MAX_MARKS = 100; → correct
}
```

→ final variables should be initialized in a class at the time of ~~declaration~~ inside the constructor.

Ex:

```
Class FinalVarEx
{
    final int MAX_MARKS;
    FinalVarEx()
    {
        MAX_MARKS = 200; ✓ correct
    }
}
```

→ Final variable are declare with as static keyword.

Ex:

```
class FinalVarEx
{
  final static int MAX_MARKS = 300;
}
```

→ final variables also declared as Local variables.

Ex:

```
class FinalVarEx
{
  public static void main (String args[])
  {
    final int y = 30; //Local variable
    S.o.pln(y);
  }
}
```

* Scope and lifetime of variable:

Scope:

→ The Scope of a variable refers to the region of the program where the variable is accessible and can be used.

Life time:

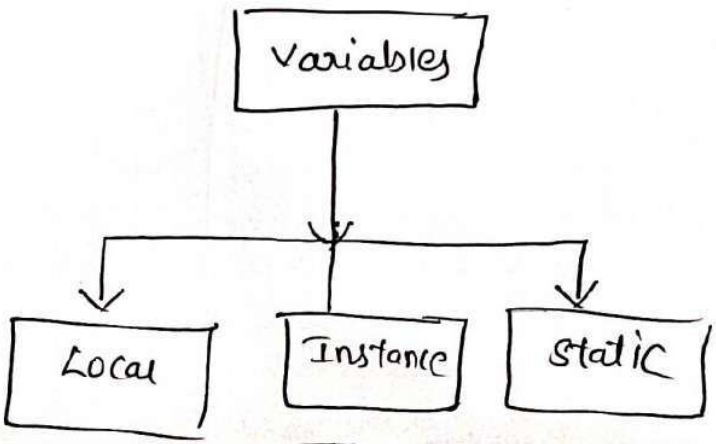
→ The Lifetime of a variable is the duration during which it exists in memory and retains it's value.

Variables are three types

(i) Local variable

(ii) instance variable

(iii) Static variable



(i) Local variables:

→ declared inside any block, method, constructor

→ Accesible with in that scope, cannot access outside.

→ gets initialized (or) created when method start Execution and destroyed when method stop Execution

Ex

```
class LocalVariableDemo
{
    static void display()
    {
        int a=5, b=10; // Local variable
        S.O.println("sum = "+(a+b));
    }
    psvm(String args[])
    {
        display();
    }
}
```

Output

Sum = 15

(ii) instance variable

→ declared inside the class, but outside the method.

→ Accessible any where within the class

→ gets initialized (or) created when the object is created and destroyed when the object is destroyed.

fx:

```
class InstanceVarDemo
{
    int a,b,c;
    float f1,f2,f3,f4;
    double d1,d2;
                        } → Instance variables

    psvm(String args[])
    {
        S.O.P("Instance variables Demo");
    }
}
```

(iii) Static variable :

→ declared inside the class, but outside the method with static keyword.

→ Accesible any where in the class

→ get initialized, when the class start execution and destroyed when the class stop execution.

Ex:

```
class StaticvarDemo
{
    int a,b,c;
    float f1,f2;
    double d1,d2;          } → Instance Variables
    Static int s1,s2;
    Static float r1,r2;    } → Static Variable

    psvm(String args[])
    {
        { int x=5, y=10, z=15; }   → Local Variable
        S.o.p("Static variable Demo");
    }
}
```

* Class and object : (Introduction)

Class :

→ The class keyword is used to declare a new java class, which is a collection of related variables and/or methods.

→ Classes are the basic building blocks of object-oriented programming.

→ A class is a template for an object.

→ Class is a blueprint of an object.

Class Declaration:

```
Class <class-name>
{
    return type variable1;
    return type variable2;        // Variables (or) Properties
        :
    return type method name1(parameters)   <-- optional
    {
        ---;  //Statements
    }
    return type methodname2
    {
        ---;  //Statements
    }
}
```

Methods (or) Actions

Object:

→ An object in java is a basic unit of object-oriented programming and represents real-life entities.

→ objects are the instances of a class.

→ object contains variables and methods.

→ object is physical entity.

Syntax:

<Class-name> <object-name> = new <Class-name>();

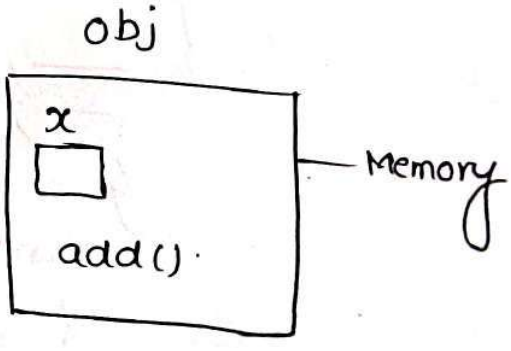
Example:

Kalam obj = new Kalam();

(used to allocate memory for the object)

↓ Keyword

```
Class Kalam
{
    int x;
    void add();
    {
        —;
        —; //stmts
    }
}
```



* Difference between class and object:

Class	object
→ class is a blueprint from which objects are created.	→ object is instance of the class.
→ class is a collection of objects.	→ object is a real world Entity such as book, pen, board, person, table, etc.
→ class is a logical Entity	→ object is a physical Entity.
→ class is defined only once.	→ objects can be created by many as per requirement.
→ class doesn't Allocate memory for variables when it is created.	→ objects Allocate memory for variables when it is created.
→ class is declared using class keyword.	→ new operator

Syntax:

```
Class <class-name>
{
  variables;
  method
}
```

```
Classname obj = new classname();
```

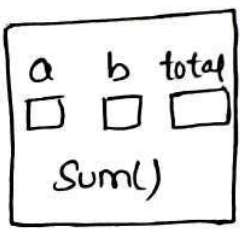
Example Program:

```
import java.lang.*;

Class Addition
{
    int a, b, total; // Instance Variable
                     (or)
                     Class variable

    void Sum()
    {
        a = 5;
        b = 10;
        total = a+b;
        System.out.print (" Addition of Two numbers = "+total);
    }
}

class Maddition
{
    public static void main (String args[])
    {
        Addition obj = new Addition();
        obj.Sum();
    }
}
```



Save : Maddition.java

Compile : javac Maddition.java

Run : java Maddition

→ Class members in java refer to the

* Class members: fields (variables) and methods (functions) that are defined with in a class.

→ All the variable declared (or) and method defined inside a class are called Class members.

Method:

→ The block in which code is written is called method (member function).

The Java programming language defines the following kinds of variables:

There are 4 types of java variables

→ instance variables
→ class variables
→ local variables
→ Parameters

instance variables:

→ Instance variables are declared inside a class. Instance variables are created when the objects are created. They take different values for each object.

Class variables:

→ class variables are also declared inside a class. (or) so these are accessed by all the objects of the same class. Only one memory location is created for a class variable.

Local variable:

→ variables which are declared and used with in a method are called local variables.

Parameters:

→ variables that are declared in the method Parenthesis are called Parameters.

Class and object Example

```
import java.lang.*;
Class Test
{
  int a;
  void SetData(int i) // method
                 └── parameter
    {
      a=i;
    }
    int displayData() // method
      {
        return a;
      }
    }
Class Access Test
  {
    psvm (String args[])
      {
        Test ob = new Test(); // object creation
          ob.SetData(50);
        S.o.pln("value of a is:-" + ob.displayData());
      }
  }
```

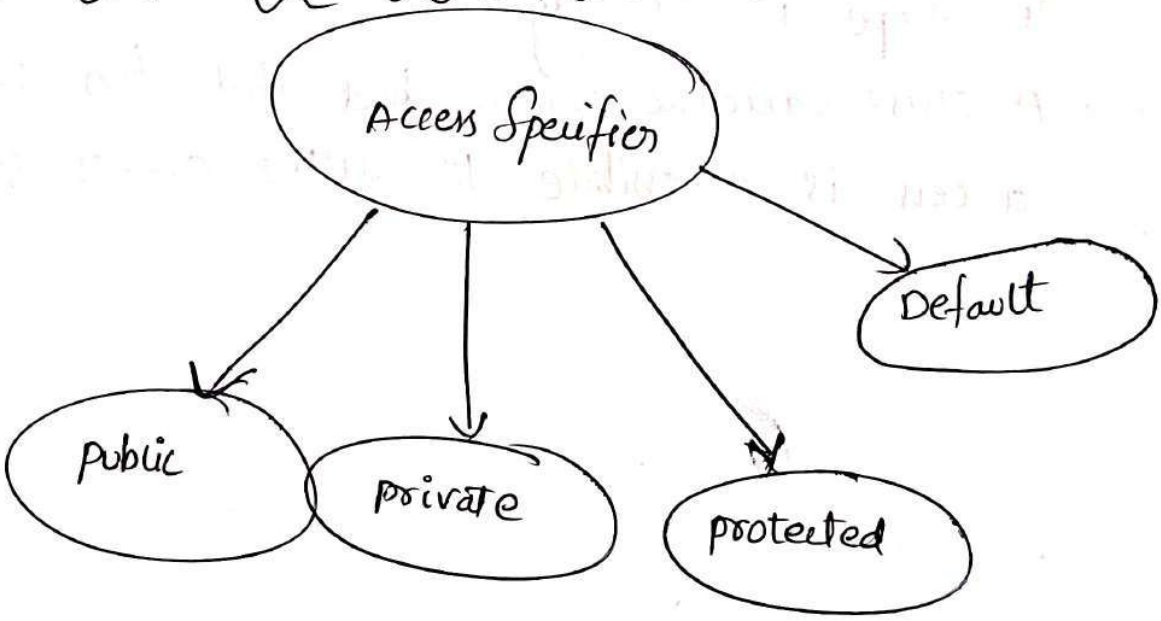
Output:

value of a is: 50

* Access Specifiers (or) Access control (or) access modifiers (or) Access control for class members (or) Accessing private members of class:

→ An access specifier determines which feature of a class (class itself, data members, methods) may be used by another class.

Java Supports four access Specifier:



public:

→ If the members of a class are declared as public then the members (variables/methods) are accessed by out side of the class.

private:

→ If the members of a class are declared as private then only the methods of same class can access

private members (variables / methods).

Protected :-

→ The protected access modifier in Java is a keyword that controls access to member variables & methods.

Default Access :

→ If the access Specifier is not Specified, then the Scope is friendly.

→ A class variable (or) method that has friendly access is accessible to all the classes of a package.

* Example:

```
import java.lang.*;
Class Test
{  int a; // default accey
   public int b; // public access
   private int c; // private access
   //methods to access c
   void setData(int i)
     {
       c = i;
     }
   int display()
     {  return c;
     }
}
Class AccessTest
{  psvm(String args())
     {
       Test ob = new Test();
       // a and b can be accessed directly
        ob.a = 10;
        ob.b = 20;
      // 'c' can not be accessed directly because it is private
       ob.c = 100; //error
       ob.setData(100);
      S.o.pln("value of a, b and cae:" + ob.a + " ", ob.dispData()); ob.b + " " + ob.dispData());
     }
}
```

* unit-2 *

G. paramesh
Asst. professor
cse-1

* Classes and objects

→ Java is an object-oriented language.

→ Java is organized in such a way that everything your program in it becomes either a class (or) an object.

→ classes and objects are closely related & work together.

Class

→ A class is a group of objects which have common properties.

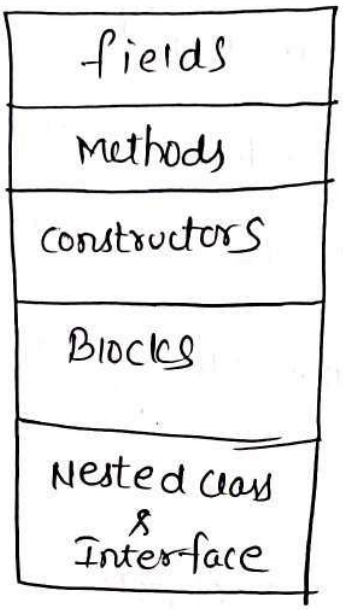
→ It is a template (or) blueprint from which objects are created. It is a logical entity. It can't be physical.

→ We can define a class as container that stores the data members and methods together. These data members & methods are common to all the objects present in a particular package.

→ In java, a class is a template that defines the structure & behaviour of objects

→ A class encapsulates data (in the form of variables) and methods (functions) that operate on that data.

Class in java



Syntax to declare a class;

```
class <class_name>
{
  variable Declaration 1;
  variable Declaration 2;
  ..... ;
  ..... ;
  variable Declaration n;
returntype methodname1 ([Parameters list])
    { //body of the method
    }
returntype method name 2([Parameters list])
    { //body of the method
    }
  } // end of class
```

* Object:

→ Object is an Entity of a class.

→ An object is an instance of a class. A class is a template (or) blueprint from which objects are created. So, an object is the instance of a class.

→ Object is a real time Entity.

→ The real-world objects have State & behaviour.

Syntax: ClassName objectName = new className();

An object in Java consists of the following:

(i) Identity
(ii) Behaviour
(iii) State

(i) Identity:

→ This is the unique name given by the user that allows it to interact with other objects in the project.

Ex:

→ Name of the Student.

(ii) Behaviour

→ The Behaviour of an object is the method that you declare inside it. This method interacts with other objects present in the project.

Ex: Studying, writing, playing

(iii) State:

→ The parameters present in an object represent its state based on the properties reflected by the parameters of other objects in the project.

Ex: Roll number, Section

Class & object Example:

```
import java.lang.*;
Class Demo
{ int a;
  void setData (int i)
    { a = i;
    }
  int displayData()
    {
      return a;
    }
  }
Class AccessDemo
  { psvm (String args[])
    {
      Demo ob = new Demo(); //object creation
      ob.setData(5);
      S.o.pln(" value of a is:"+ob.displayData());
    }
  }
```

Output

value of a is: 5

* Constructor:

→ It is a special type of method.

→ It is used to initialize an instance variable.

→ Name must be same as class Name.

→ It can not contain return type.

→ It will execute during object creation time.

→ It will execute one time for every object created.

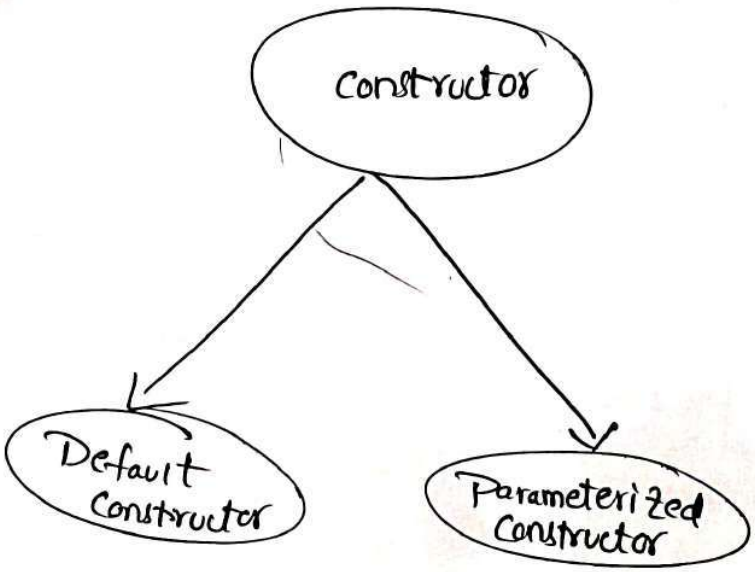
→ It can contain any number of parameters.

(i) Zero parameterized Constructor (or) Default Constructer

(ii) parameterized Constructor

→ A class can contain any number of constructor

There are two types of Constructors.



(i) Default Constructor:

→ A constructor which does not accept any parameter is called "default constructor".

(ii) parameterized Constructor:

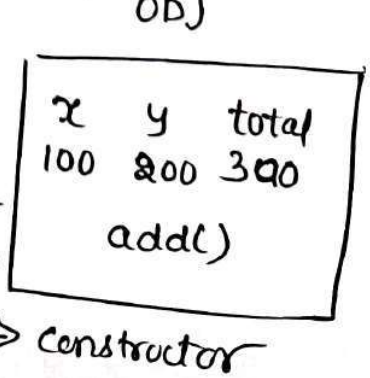
→ A constructor that accepts arguments is called parameterized constructor.

// Constructor Example

```
import java.lang.*;

Class DemoConstructor
{
  int x, y, total; // Instance Variables

   DemoConstructor() // Default constructor
    {
      x=100;
      y=200;
    }
  Void add() //method
    {
      total = x+y;
      S.O.pln ("Sum =" + total);
    }
}
class Mdemo
{
  psvm(String args[])
    {
DemoConstructor obj= new DemoCons-
                     tructor();
      obj.add();
    }
}
```



Output

Sum = 300

//A Class contain any number of constructors

Example

```
import java.lang.*;

Class Display
{
  psvm(String args[])
   {
   Display d1 =new Display();
   Display d2 =new Display();
   Display d3 = new Display();
  }
  Display() // Constructor
  {
    S.o.pln(" Kalam Sir");
   }
}
```

output:

Kalam Sir
Kalam Sir
Kalam Sir

* Default Constructor:

→ A constructor without arguments (or) without parameters is called "Default Constructor."

Rules:

→ Same as classname

→ No returntype

→ implicit calling

Syntax:

```
classname( )
{
   ___;
   ___; //Statements
   ___;
}
```

* // Default constructor program

```
import java.lang.*;

Class DefDemo
{
    int a, b, total;

    DefDemo() // Default Constructor
    {
        a = 100;
        b = 200;
    }
    void add() //method
    {
        total = a+b;
        S.o.pln("Sum = "+(a+b));
    }
}

Class Demo
{
psvm(String args[])
    {
    DefDemo obj = new DefDemo();
        obj.add();
    }
}
```

obj

a	b	total
100	200	300

add()

DefDemo() → Default constructor

Output

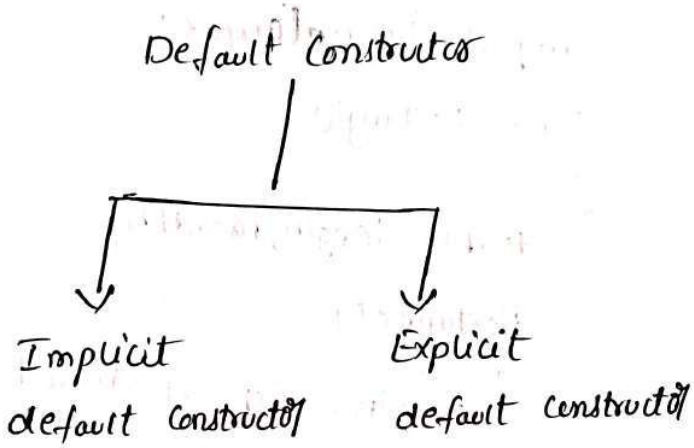
Sum = 300

→ Default constructor

↳ It is of 2 types

(i) Implicit default Constructor

(ii) Explicit default constructor



(i) Implicit default constructor:

```
Class Rectangle
  {
     double length, breadth;
  }
  class Constructor
  {
     psvm (String args[])
        {
           Rectangle r1 = new Rectangle();
        S.o.pln ("length is" + r1.length);
        S.o.pln ("breadth is" + r1.breadth);
        }
  }
```

output

length is 0.0
breadth is 0.0

Note:

→ If the default constructor is not explicitly defined, then System default Constructor automatically initializes all instance variables to zero.

(ii) Explicit Default Constructor:

```
import java.lang.*;
class Rectangle
{
  double length, breadth;
   Rectangle()
   { S.o.pln ("Explicit default Constructor");
     length = 10.0;
     breadth = 20.0;
   }}
class Constructor
  { psvm (String args[])
      {
        Rectangle r1 = new Rectangle();
S.o.pln ("length is "+ r1.length);
S.o.pln (" breadth is "+r1.breadth);
      }
  }
```

Output

```
Explicit default Constructor
length is 10.0
breadth is 20.0
```

3) **Parameterized Constructor:** A constructor that accepts arguments is called parameterized constructor.

```
* import java.lang.*;

Class Student
  {
    int rollno;
    String name;
    String branch;
     psvm(String args[])
      {
        Student s1 = new Student(101, "vaishnav");
        Student s1 = new Student(102, "vaibhav", "CSE");
        Student s1 = new Student("CSE");
      }
    Student (int x, String n)
     {
       rollno = x;
       name = n;
       S.o.pln ("NAME:" + name + "ROLLNO:" + rollno);
      }
      Student (String b)
        {
          branch = b;
          S.o.pln (branch);
        }
      Student (int x, String n, String b)
        {
          rollno = x;
          name = n;
          branch = b;
          S.o.pln(name + " -->" + rollno + "-->" + branch);
        }
    }
```

output

```
101  vaishnav
102  vaibhav  CSE
CSE
```

* Constructor overloaded:

→ writing more than one constructor with in a same class with different parameters is called constructor-overloading.

(i) No of parameters
(ii) Order of parameters
(iii) Type of parameters

Example

```
Add( )
{
  —;
  —;
}
Add ( int x)
{ —;
  —;
}
Add (float x)
{ —;
  —;
}
```

1 parameter match type different

```
Add( )
{
  —;
  —;
}
Add ( int x)
{
  —;
  —;
}
Add ( int y)
{ —;
  —
}
```

parameter match type match ↓ NO Constructor overloading concept here

```
Add( int x, float y)
{
  —;
  —;
}
```

— Here order different (apply Constructor overloading.)

```
Add (float x, int y)
{
  —;
  —;
}
```

// Example Constructor overloading

```
import java.lang.*;
Class Add
{
  int x, y, total;
  Add( ) // Zero Argument
  { x=10;
    y=20;
  }
  Add( int x)//One Argument
  {
    x=a;
    y=a;
  }
  Add (int a, int b) // Two Argument
  { x=a;
    y=b;
  }
  void Sum()
  {
    Total = x+y;
    S.o.pln ("Sum is:" + total);
  }
}
Class Madd
{
  psvm( String args[])
  {
    Add ob1 = new Add();
    Add ob2 = new Add(30);
    Add ob3 = new Add (40,50);
    ob1.Sum(); ob2.Sum(); ob3.Sum();}}}
```

ob1

x	y	tot
10	20	

Sum()

ob2

x	y	tot
30	30	

Sum()

x	y	tot
40	50	90

Sum()

* Methods in Java:

→ Group of Statements worked together to perform an Operation is called "Method".

Syntax:

```
Modifier returntype methodname (parameters)
{
    Statements;
    ......;
    ......;
}
```

Example: // without return type & without parameter

```
import java.lang.*;

class Addition
{
Void add() // Instance method - without return type & without parameter
{
    int x=10;
    int y=20;
    int z=x+y;
    S.o.pln(z);
}
    Psvm(String args[])
    {
        Addition obj = new Addition();
        obj.add();
    }
}
```

output:

30

// with returntype and without parameter.

```
Class Addition
{
    int add()
    {
        int x=10;
        int y =20;
        int z = x+y;
        return z;
    }
    Psvm (String args [])
    {
        Addition ob = new Addition();
        int result = ob.add();
        S.o.pln (result);
    }
}
```

Output:

30

//without returntype and with parameter

```
import java.lang.*;
Class Addition
  {
     void add (int x, int y)
       {
          int z = x+y;
          S.o.pln (z);
       }
     psvm (String args[])
       {
Addition ob1 = new Addition();
          ob1.add (5,10);
       }
  }
```

output

15

// with returntype and with parameter

```
import java.lang.*;

Class Addition
{
    int add (int x, int y)
    {
        int z = x+y;
        return z;
    }
    Psvm (String args[])
    {
    Addition ad = new Addition();
int result= ad.add(5,10);
    S.o.pln (*result)*
    }
}
```

Output:

// with return type & with parameter

```
import java.lang.*;
Class Addition
{
    int add (int x, int y)
    {
        int z = x+y;
        return z;
    }
    Psvm(String args[])
    {
        Addition ob = new Addition();
        int result = ob.add(5,10);
    }
}
```

Output :

15

* Nesting Methods:

→ A method can be called by using only its name by another method of the same class that is called Nesting methods.

Syntax:

```
class Main
  {
    method1()
      {
        //statements
      }
    method2()
      {
        //Statements
        //Calling method1 from method2()
        method1();
      }
    method3()
      {
        //Statements
        //Calling method2() from method3()
        method2();
      }
  }
```

*Example

```
import java.lang.*;

public class NestingMethod
 {
   public void a1()
     {
    S.o.pln ("*** Inside a1 method ***");
     a2();
   }
  public void a2()
   {
    S.o.pln ("*** Inside a2 method ***");
    }
  Public void a3()
   { S.o.pln("*** Inside a1 method ***");
     a1();
    }
  PSvm(String args[])
     {
       //Creating the object of class
  NestingMethod n = new NestingMethod();
  //Calling method a3() from main() method
   n.a3(a,b);
    }  }
```

* Assigning one object to Another (or) Cloning of objects

→ we can copy the values of one object to another using many ways like:

(1) using clone() method of an object class.

(2) using constructor

(3) By assigning the values of one object to another.

(4) In this example, we copy the vaues of an object to another by assigning the values of one object to another.

Ex:

```
Class Copy
{
    int a=100;
}
Class Copyobject
{
    psvm(String args[])
    {
        copy c1 = new copy();
        copy c2 = c1;
        S.o.pln("object c1 value:" +c1.a);
        S.o.pln("object c2 value:" +c2.a);
    }
}
```

* passing Arguments by value and by Reference:

→ There is only call by value in java, not call by reference but we can pass non-primitive datatype to function to see the changes done by callee function in caller function.

→ If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Example:
// passing primitive datatype to function

```
Class Example
{
   int a=10;
   void change(int a) // called (or) callee function
    { a = a+100;
      }
    }
   Class CallByValue
  {
    psvm(String args[]) //calling function
     {
```

```
Example e = new Example();
S.o.pln("a value before calling change(); "+e.a); //10
e.change(10); //call by value (passing primitive datatype)
S.o.pln ("a value after calling change(): "+e.a); //10
}
}
```

* Method overloading:-

→ If a class has multiple methods having same name but different in parameters. It is known as Method overloading.

Example:

```
import java.lang.*;
Class Method
{
   int add(int a,int b)
    { S.o.pln(" Iam Integer method ");
      }return a+b;
   float add(float a,float b, float c)
   {
     S.opln(" Iam float method ");
     return a+b+c;
   }
  int add(int a,int b, int c)
    {
      S.o.pln(" Iam Integer method");
      return a+b+c;
    }
       float add(float a, float b)
          { S.o.pln("Iam float method");
            return a+b; } }
```

```
Class MethodOverLoad
{
  psvm(String args[])
  {
    Method m = new Method();
    S.o.pln (m.add(10,20));
    S.o.pln (m.add(10.2f, 20.4f, 30.5f));
    S.o.pln (m.add (10,20,30));
    S.o.pln (m.add (10.2f, 20.4f));
  }
}
```

* Overriding Method:

→ If Subclass has the same method as declared in the parent class, it is known as method overriding.

Uses: Runtime polymorphism

Rules:

→ There must be a inheritance.

→ The method must have same method signature as in the parent class, it is known as method overriding.

Example:

```
import java.lang.*;
    Class SuperClass
    {
        void Calculate(double x)
        { s.o.pln("Square Value of x is:"+(x*x));
        }
    }
Class Subclass extends Superclass
    {
        void Calculate(double x)
        {
            super();
            s.o.pln("Square root of x is:"Math.Sqrt(x));
        }
    }
```

```
class MethodOver
{
    psvm(String args[])
    {
        SubClass s = new SubClass();
        s.calculate(2.5);
    }
}
```

Note:

→ When a Super class method is overridden by the Subclass method calls only the sub class method and never calls the Super class method. We can say that sub class method is replacing Super class method.

* Nested classes (or) Inner classes

→ The classes that are declared inside another class are called as Nested class (or) Inner class

(i) Nested classes have a special type of relationship that it can access all the members of outer class including private.

(ii) More readable and maintainable.

(iii) Nested class requires less code

(iv) If a class is useful to only one another class then it is logical to embed it in that class and keep the two together.

Examples

Syntax:
```
class outer_class
{
    // code
    class Inner_class
    {
        // code
    }
}
```

```
// Example - Nested classes
    Class Outer    // outer class
    {
      class Inner // Nested class (or) Inner class
      {
        void InnerMethod() // Instance Method
        {
          S.o.pln (" Nested class Method");
        }
      }
      void OuterMethod() // Instance Method
      {
        S.o.pln("outer class Method");
      }
      Psvm (String args[])
      {

      Outer ob= new Outer();
          ob.outerMethod();
      Inner ob1= new Inner(); // Here we cannot create the object for Inner class (or) Nested class
          ob1. innerMethod;
      }
    }
```

Example - Nested Class

[∵ static method to Inner class calling]

```
class Outer
{
    class Inner
    {   void innerMethod()
        {   s.o.pln(" inner class method");
        }
    }
    void outerMethod()
    {   s.o.pln("outer class method");
    }
    psvm(String args[])
    {
        Outer ob1 = new outer();
            ob1.outer method();
    Outer.Inner i = new outer().new Inner();
            i.innerMethod();
    }
}
```

output:

Outer class Method
Inner class Method

Example - Nested class

[∵ Static method to Inner class calling]

```
class Outer
{
    class Inner
    {
        void innerMethod()
        {
            s.o.pln(" inner class method");
        }
    }
    void outerMethod()
    {
        s.o.pln("outer class method");
    }
    psvm(String args[])
    {
        Outer ob1 = new outer();
        ob1.outer method();
    Outer.Inner i = new outer().new Inner();
        i.innerMethod();
    }
}
```

output:

Outer class Method
Inner class Method

// Example (Nested class)

```
Class Outer   //outer class        [∴ Instance method to Inner class calling]
{
  class Inner // Inner class
  {
    void innerMethod()  // Instance Method (Inner class)
    { s.o.pln("inner class Method");
    }
  }
  void outerMethod()  // Instance Method (outer class)
  {
    s.o.pln("outer class Method");
    Inner ob1 = new Inner();
        ob1.innerMethod();
  }
  psvm(String args[])
  {
    Outer ob = new Outer();
        ob.OuterMethod();
  }
}
```

→ javac outer.java
→ java outer

Output

outer class Method
inner class Method

* Class objects as parameters in methods:

Example:

```
import java.lang.*;
Class Example
{  int a=10;
   void change(Example x) // called (or) Callee function
    {  x.a = x.a+100;
    }
}
class CallByValue
   {  psvm(String args[])
      {
Example e = new Example();
S.o.pln("a value before calling change(): "+e.a); //10
    e.change(e); // Call by value (passing non
                    primitive datatype) (or)
                    class object as parameters in
                    a method
S.o.pln("a value after calling change(): "+e.a); //110
    }
}
```

* Recursion method:

→ Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.

Syntax

```
returntype methodname()
  {
   //Code to be executed
   method name(); //Calling same method
  }
```

Example:

```
import java.lang.*;
class Factorial
 { int fact(int n)
    { if(n==1)
       return 1;
      int x = n*fact(n-1);
      return x;
    }
   psvm(String args[])
    {
  Factorial f = new Factorial();
  int result = f.fact(5);
    S.o.pln(result); } }
```

Output

120

Advantages of Recursion:

→ It can make your code easier to write.
→ It can make your code efficient
→ It can reduce the amount of time it takes your solutions to run..
→ Recursion is efficient at traversing tree Data Structure.

Dis-advantages of Recursion:

→ Recursion uses more memory.
→ Recursion can cause Stack overflows.
→ Recursion can be slow, if you don't use it's correctly with memoization.
→ Recursion can be confusing

* Attributes of Static & final:-

Static:

→ Indicates that a variable (or) method is part of the class, not the object.

→ Static variables can be reinitialized and don't need to be initialized when declared.

→ Static is used for memory management and can be applied to variables, methods, blocks and nested classes.

Final:

→ Declares a variable as constant and prevents the user from accessing a method, variable (or) class.

→ Final variables must be initialized when declared and cannot be reinitialized.

Here are some other things to know about the Static & final keywords:

Global Constants:

→ Static & final are used to create global constants in Java.

Inheritance:

→ Final classes cannot be inherited by any sub-class.

Security:

→ Final methods can be used to maintain security and consistency by preventing subclasses from altering critical methods.

Single assignment:

→ The final keyword makes a variable a single-assignment variable, meaning that the compiler expects exactly one assignment to that variable.

~~[illegible] between Static & Final~~

* Static variable & Static methods :

→ The Static keyword in java is used for memory management. It makes your program memory efficient [i.e it saves memory].

→ It can apply for variables, methods, blocks.

→ The Static variable can be used to refer the common property of all objects.

→ Static variables gets memory only once in class area at the time of loading.

* Program: // Display Student Details

```
import java.lang.*;
class Student
{
    int rollno; // Instance Variable
    String name; // Instance Variable
    Static String college = "Aditya"; // Static Variable

    Student(int r, String n) // Constructor
    {
        rollno = r;
        name = n;
    }
    void display() // Instance Method
    {
        S.o.pln(rollno + " " + name + " " + college);
    }
    public Static void main (String[] args)
    {
        Student st1 = new Student(500, "vaibhav");
        Student st2 = new Student(501, "vaishnav");
        st1.display();
        st2.display();
    }
}
```

Output:

```
500  vaibhav  Aditya
501  vaishnav  Aditya
```

* this: (Keyword)

→ This keyword refers to current instance of a class.

→ Using this keyword we can access instance members [variables & methods].

→ Specifying this keyword is sometimes optional and sometimes mandatory.

this:

Mandatory: If there is a confusion b/w instance variable and local variable then specifying this keyword is mandatory. In such case we don't specify this keyword then the compiler will also not specify this keyword and instead access local variable. If we want to access instance variable then the programmer has to specify this keyword.

Program2:

```
import java.lang.*;
class Demo1
{
    int z = 300;
}
class Demo extends Demo1
{
    int x = 200;
    void m1()
    {
        int x = 100;
        S.o.pln(x);
        S.o.pln(this.x);
        S.o.pln(z);
    }
    psvm(String args())
    {
        Demo d = new Demo();
        a.m1();
    }
}
```

op: 100
200
300

* Final class: (or) Inhibiting Inheritance of class using Final:

→ The final class is a class that is declared with the final keyword.

→ Sub classes can't inherit a final class, and any Subclass cannot inherit a final class.

→ So, we can restrict class inheritance by using a final class. Ex import java.lang.*;

```
final class Demo1
{
    void display()
    {
      S.o.pln ("This is final class");
    }
}
class Demo2
{
    psvm(String args[])
    {
        Demo1 ob = new Demo1();
        ob.display();
    }
}
```

Syntax

Example

```
final class A
{ ---;
  ---;
}
Class B extends A // cannot be inherited
{ ---;
  ---;
}
```

* Final method:

→ A final method in java is a method that cannot be overridden by any Sub class.

→ The final keyword is used in the method declaration to indicate that the method is final.

Example

```
import java.lang.*;
Class Demo1
{
  final void display()
  {
    S.o.pln("This is final method");
  }
}
Class Demo2 extends Demo1
{
  void display1()
  {
    S.o.pln("This is Sub-class method");
  }
}
Class Mfinal
{ psvm(String args[])
  { Demo2 ob = new Demo2();
    ob.display();
    ob.display1(); } }
```

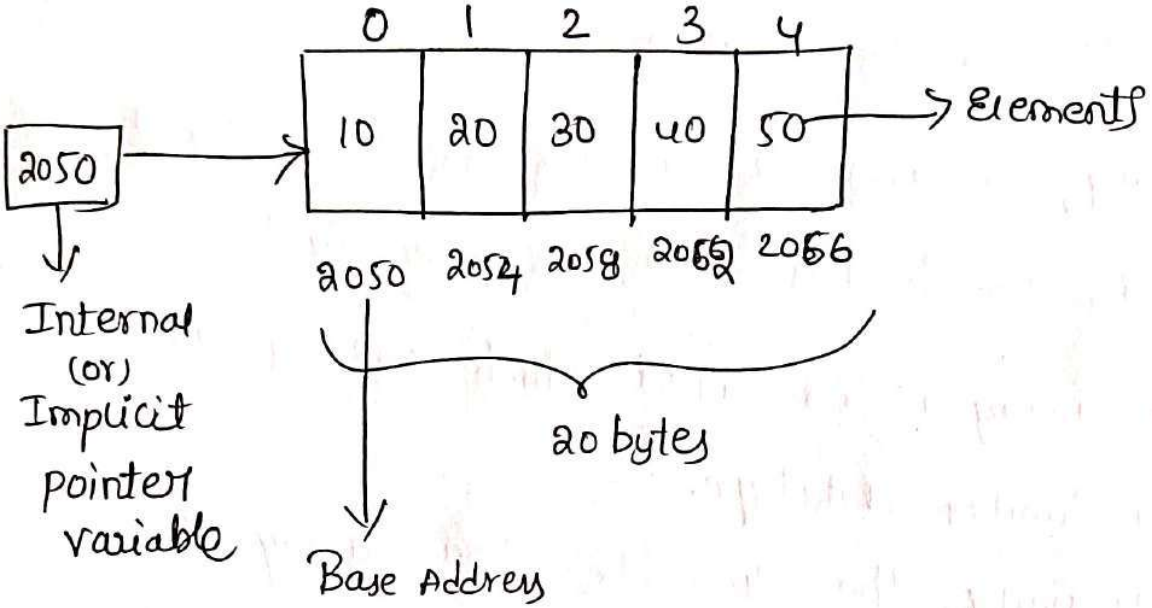
Output

```
This is final method
This is Sub-classmethod
```

Ex: Datatype arrayname[] = new Datatype [size];

int arr[] = new int [5];

→ int arr[] = {10, 20, 30, 40, 50}; → Index




```
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
```

Advantages:

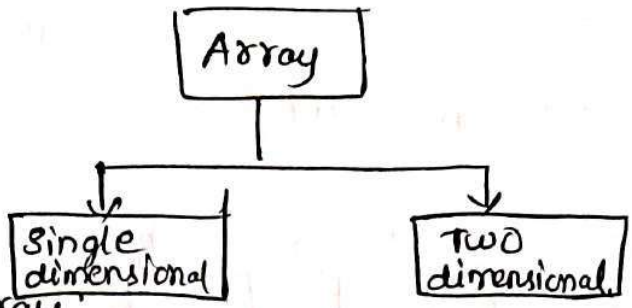
→ Code optimization. It makes the code optimized. we can retrieve or sort the data efficiently.

→ Random access: we can get any data located at an index position.

Dis-advantages:

→ Size Limit: we can store only the fixed size of elements in the array. It doesn't grow its size at runtime.

* Types of Array:



One-dimensional Array:

→ Collection of x Elements (or) items which have one Similar type variable name & one subscript is called 1Dimensional Array.

→ Subscript

int x[]; → 1D-array

int x[][]; → 2D-array

int x[][][]...; multidimensional array

(i) Declaration:

Syntax:

Datatype arrayname[];

Ex: int arr[];

(ii) Initialization:

Syntax:

arrayname = new datatype[size];

Ex:

arr = new int[5];

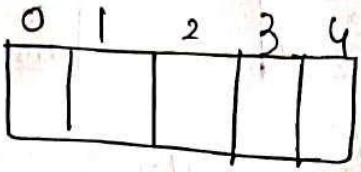
Combination (Initialization + Declaration)

Syntax:

Datatype array-name[] = new datatype[size];

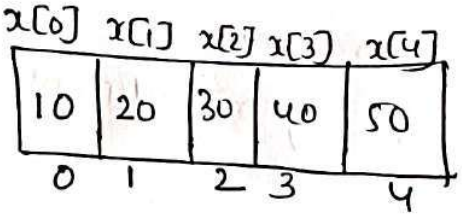
Ex:

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



Example:

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

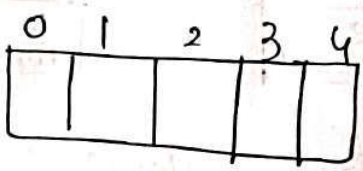



```
x[0] = 10;
x[1] = 20;
x[2] = 30;
x[3] = 40;
x[4] = 50;
```

Generous ...
runtime.

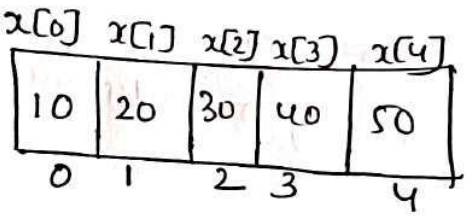
Ex:

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



Example:

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




```
x[0] = 10;
x[1] = 20;
x[2] = 30;
x[3] = 40;
x[4] = 50;
```

Generous ...
runtime.

* Two Dimensional Array:

→ A two-dimensional Array, also known as a 2D array, is a collection of data Elements arranged in a grid-like structure with rows and columns.

Declaration

Syntax

```
returntype array_name [][];
```

Ex:

```
int arr[][];
```

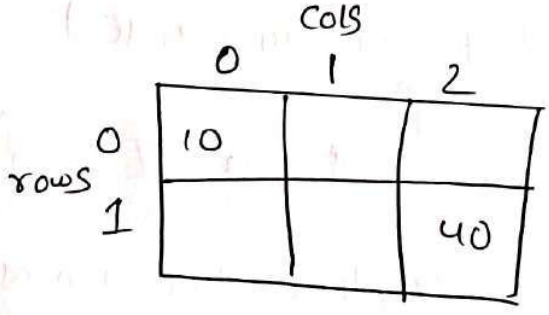
Construction of an array:

Syntax:

```
array_name = new datatype [rowsize] [column size];
```

Ex:

```
arr = new int [2][3];
```



Intialization:

Syntax

```
arrayname [row pos] [col pos] = value;
```

Ex:

```
arr[0][0] = 10;
arr[1][2] = 40
```

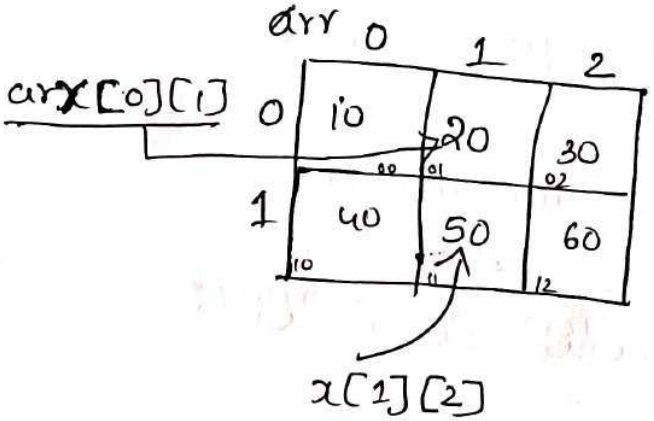
Combination of (Intialization + Declaration)

Syntax:

returntype arrayname [] [] = { list of values };

Ex:

int arr[] [] = { {10,20,30}, {40,50,60} };



* Example: Two Dimensional Array

```
import java.lang.*;
Class DemoTwoDim
{ psvm (String args[])
  {
    int arr[][] = { {10,20,30}, {40,50,60} };
  S.o.pln ("Two Dimensional Array Elements");
  for(r=0 ; r<=1 ; r++) //rows
  { for ( c=0; c<=2 ; c++) //cols
    { S.o.p( arr [r][c]);
    } }
  S.o.pln(); }
```

Output

```
10 20 30
40 50 60
```

Example:

```
import java.lang.*;
class ArrayDemo
{
    psvm (String args[])
    {
        int arr[] = new int[5];
        arr[0] = 5;
        arr[1] = 3;
        arr[2] = 4;
        arr[3] = 9;
        arr[4] = 6;
        for(int x:arr)
        {
            S.O.pln(x);
        }
    }
}
```

output

5
3
4
9
6

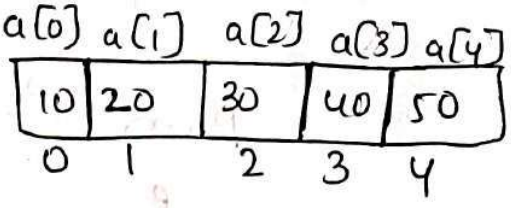
* Example (One Dimensional Array)

```
import java.lang.*;

class Demooneoim
{
  psvm (String args[])
    {
      int a[] = new int [5];

        a[0] =10;
        a[1] =20;
        a[2] =30;
        a[3]=40;
        a[4]=50;
     S.o.pln (" 1D array elements are : ");

  for (int  i=0; i<=4; i++)
     {
        S.o.pln (a[i]);
        }
     }
}
```



output

```
10
20
30
40
50
```

* Declaration and Initialization of Single dimensional Arrays:

Syntax of Declaration an Array:

Syntax:

```
datatype variableName[size];
        (or)
datatype[size] variableName
        (or)
datatype [] variableName;
```

Ex: `int arr[5];`

(or)

`int[5] arr;`

(or)

`int[10] arr;`

Note:

→ The array arr is initially set to null. new is a special operator the allocate memory.

Initialization of an Array in Java:

Syntax:

```
array variableName = new datatype[size];
```

Example

```
int arr[5]; // declaration

Arr = new int[5]; // initialization of array
                     without values
```

(or)

```
int arr[] = new int[5]; // declaration & instantiation
```

intialization of array without values in single line.

Initialization of an Array with values:

```
int arr[] = new int[5]{10,20,30,40,50};
```

(or)

```
int arr[] = {10,20,30,40,50};
```

Note:

→ If the array is initialized at the time of declaration then new keyword is not required.

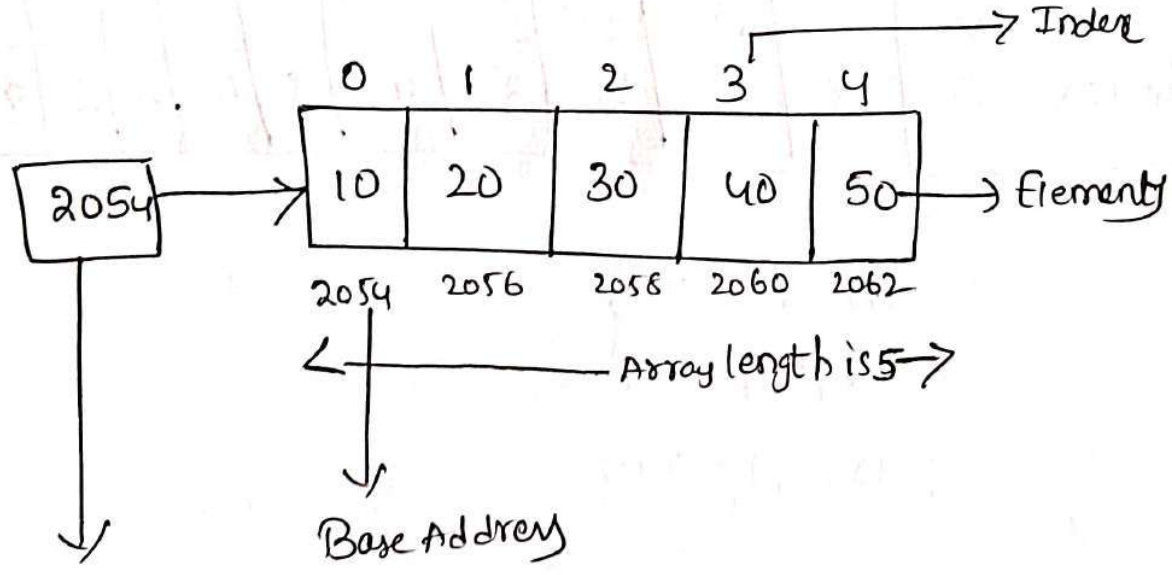
* Storage of Array in Computer Memory:

→ The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements.

→ Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index & so on and index ends with array-size-1.

Ex:

```
int arr[] = {10,20,30,40,50};
```




```
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
```

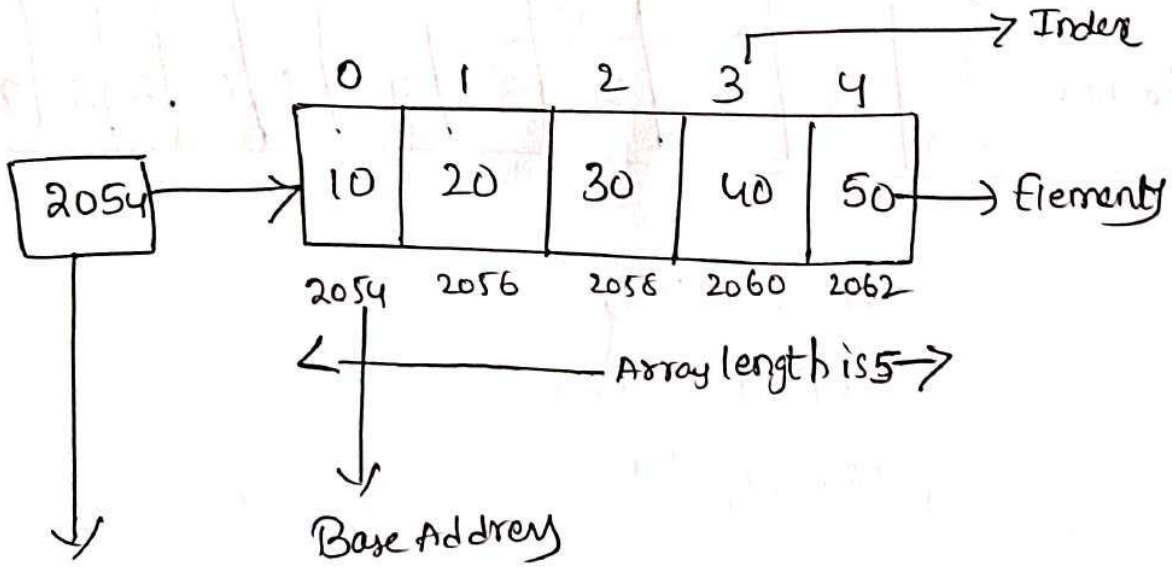
* Storage of Array in Computer Memory:

→ The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements.

→ Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index & so on and index ends with array-size-1.

Ex:

```
int arr[] = {10,20,30,40,50};
```




```
arr[0] = 10;
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
```

* Accessing Elements Of Arrays:-

→ Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index & so on & index ends with array-size-1.

→ With help of arrayName[index] syntax we can access individual element of an array.

Example

arrayName = ar	10	20	30	40	50	60	70	80	90	10
index	0	1	2	3	4	5	6	7	8	9

```
ar[3] = 40;
ar[6] = 70;
ar[9] = 100;
```

* Assigning Array to Another Array:

→ It is nothing but copying one array to another array.

Example

Assigning (copying) one Array to another Array:

```
import java.util.Scanner;
class ArrayCopy
{ psvm (String args[])
  {
int a1[] = new int[10];
int a2[] = new int [a1.length];
Scanner s = new Scanner(System.in);
for(int i=0; i< a1.length ; i++)
  {
    s.o.pln(" Enter value at index "+i);
     a1[i] = s.nextInt();
  }
for (int i=0; i<a1.length; i++)
  {
     a2[i] = a1[i];
  }
```

```
S.o.pln (" Array1 Elements: ");
for (int element : a1)
 {  S.o.pln( element + "\t");
    }
 S.o.pln (" ");
 S.o.pln (" Array 2 Element: ");
 for(int element : a2)
     {  S.o.pln(element + "\t");
 } } }
 }
```

* Dynamic change of ArraySize / dynamic Array :

→ An array is a fixed size, homogenous data Structure. The limitation of arrays is that they're fixed in size. It means that we must specify the number of elements while declaring the array.

→ The dynamic array is a variable size list data Structure. It grows automatically when we try to insert an element if there is no more space left for the new element.

→ It allows us to add and remove elements. It allocates memory at runtime using the heap. It can change its size during runtime.

→ In Java, ArrayList is a resizable implementation. It implements the List interface & provides all methods related to the list operations.

Example

```
import java.util.ArrayList;
import java.util.Scanner;

class ArrayDynamic
{
  psvm(String args[])
  {
ArrayList<Integer> a = new ArrayList<Integer>();
Scanner s = new Scanner(System.in);
char ch;
do
{ S.o.pln("** Dynamic Array **");
 S.o.pln("choose any option");
 S.o.pln("1. insert element into array");
 S.o.pln("2. remove element from array");
 S.o.pln("3. clear all element of an array");
 S.o.pln("4. do you want to know size");
 S.o.pln("5. print element of the dynamic array");
 int option = S.nextInt();
   Switch(option)
   {
      Case 1 : S.o.pln("Enter element to insert:");
               int ele = S.nextInt();
                a.add(ele);
               break;
```

```
case 2: S.o.pln ("Enter index of the element to remove:");
        int index = S.nextInt();
            a.removeCindex);
            break.
case 3: a.clear(); break;
case 4: S.o.pln ("size of the array is -" + a.size());
        break;
case 5: S.o.pln (" Element of the dynamic array:");
        S.o.pln (a);
        break;
default: S.o.pln (" Invaid option choosen");
        break;
    }
S.o.pln ("do you want to continue (Y/y (or) N/n)");
ch = S.next. charAt(0);
} while(ch == 'Y' || ch == 'y');
    }
}
```

* Search for values in Arrays:
  - ↳ write a binary Search program (Lab program)

* Sorting of Arrays :

// Sorting of an array in ascending & descending order without using Sort() method.

```
import java.lang.*;
Class SortingAscDes
{
  psvm (String args[])
  {
    int a[]= {10,345,2,789,45,34};
S.o.pln (" Array Before Swapping ");
for(int element : a)
    S.o.pln (element + "\t");
for (int i=0; i<a.length; i++)
{
  for(int j= i+1; j<a.length; j++)
    {  if (a[i] >a[j])
        {  int temp =a[i];
           a[i] =a[j];
           a[j] =temp;
        }
    }
}
```

```
S.o.pln("\n Array After Swapping in Ascending order:");
 for (int element : a)
   S.o.p(element + "\t");
for (int i=0; i<a.length; i++)
   {
      for(int j= i+1; j<a.length; j++)
        {
          if (a[i] <a[j])
            {
              int temp =a[i];
               a[i] = a[j];
               a[j] = temp;
            }
        }
      S.o.pln("\n Array After Swapping in
                descending order:");
      for (int element : a)
       S.o.p(element + "\t");
   }
```

* Class Arrays:

→ The java.util.Arrays class contains a static factory that allows arrays to be viewed as lists.

Following are the important points about Arrays

→ This class contains various methods for manipulating arrays (Such as Sorting & Searching).

→ The methods in this class throw a NullPointer-Exception if the specified array reference is null.

Array class methods:

→ asList()
→ binarySearch()
→ copyOf()
→ deepEquals()
→ equals()
→ hashCode()
→ fill()
→ sort()
→ toString

Example

↓

Example: asList() method

```
import java.util.Arrays;
import java.util.List;
Class ArrayDemo
  {
  psvm(String args[])
    {
     //create an array of Strings
  String a[] = { "abc", "klm", "xyz", "pqr" };
  List list1 = Array.asList(a);
   //printing the list
  s.o.pln ("The list is: " + list);
    } }
```

* Arrays of varying Lengths;

```
import java.util.Scanner;
Class ArrayVarying
{
  psvm(String args[])
  {
    s.o.pln("Enter the size of the array:");
Scanner S = new Scanner(System.in);
  int n = S.nextInt();
  int a[] = new int[n];
s.o.pln("Enter the "+n+"array elements");
  for(int i=0; i<a.length; i++)
    {
      s.o.pln("Enter element at index-"+i);
        a[i] = S.nextInt();
      }
      s.o.pln("array contain following elements");
      for(int ele:a)
          s.o.pln(ele+"\t");
    }
}
```

* Three dimensional Arrays:

→ An array with three indexes (subscripts) is called three dimensional array in java.

→ In other words, a three dimensional array is a collection of one (or) more two-dimensional arrays, all of which share a common name.

Example

```
import java.util.Scanner;

Class Three
{ psvm (String args[])
    {
Scanner s = new Scanner (System.in);

int a[][][] = new int [3] [3] [3]; // 3-D array declaration

for (int i=0; i<3; i++)
  {
    for (int k=0; k<3; k++)
      {
        s.o.println(" Enter element at index : "+ i+j+k);
        a[i] [j] [k] = s.nextInt();
      }
  }
}
```

```
for(int i=0; i<3; i++) // printing data of 3-Darray *
{
for(int j=0;j<3;j++)
{
for(int k=0; k<3; k++)
{
s.o.pln(a[i][j][k]+"\t");
}
}
}
}
```

Java

Array as vector:

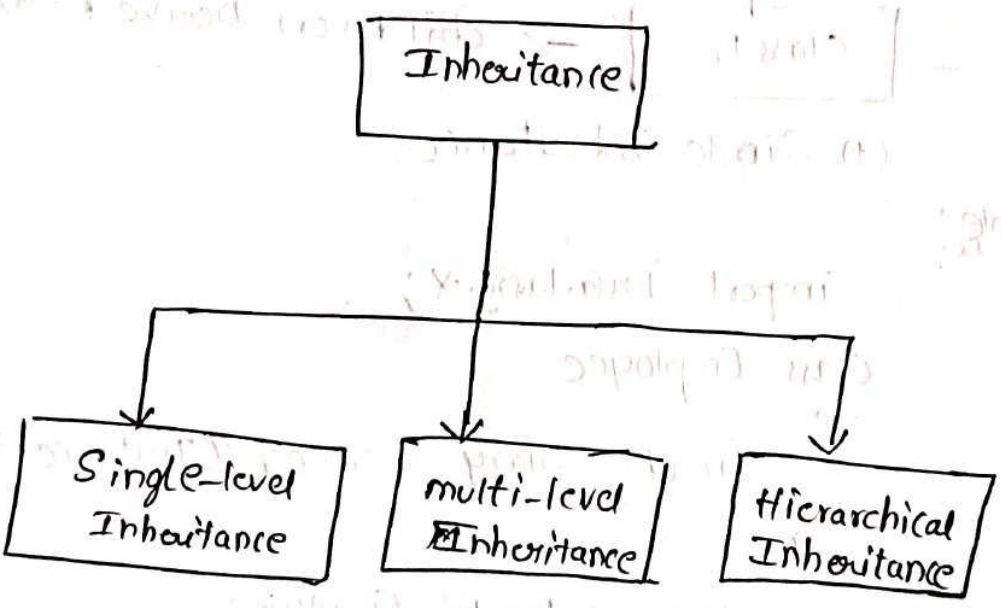
Arrays	Vector
→ The length of an array is fixed once it is created, and elements cannot be added (or) removed before its creation	→ A vector is a resizable-array that works by reallocating storage & copying the old array elements into new array.
→ In arrays are retrieval & assignment operations will be fast.	→ Vector is retrieval slow.
→ An array does not reserve any additional storage.	→ An vector reserve any additional storage.
→ An array has length property that stores its length.	→ To find the size of the vector, we can call its size() method
→ Arrays in java supports single dimensional array as well as multidimensional array.	→ A vector has no concept of dimensions, but we can easily construct a vector of vectors.

* Inheritance: (Introduction)

→ A process of acquiring the members of one class to another class is called inheritance.

→ Using inheritance we can achieve reusability and there by reduce the code size and development time of the application.

→ Java supports 3 types of inheritance they are



(i) Single level

(ii) multi level

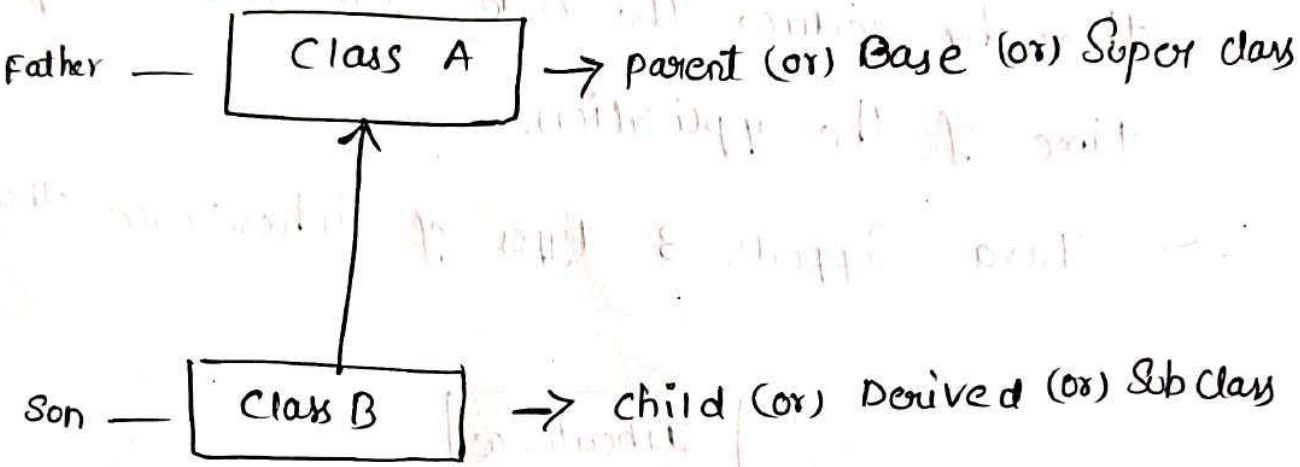
(iii) Hierarchical

Syntax

```
class <Sub class name> extends <Super class name>
{
    body
}
```

(i) Single level Inheritance:

→ The inheritance in which a single derived class is inherited from a single base class is known as "single inheritance."



(1) Single Inheritance

Example:

```
import java.lang.*;
Class Employee
{
    float Salary = 30000; // Instance Variable
}
class clerk extends Employee
{
    float bonus = 10000;
    psvm (String args[])
    {
        clerk c = new clerk()
        S.o.pln ("Total Salary:" + (c.Salary + c.bonus))
    }
}
```

Save: Clerk.java

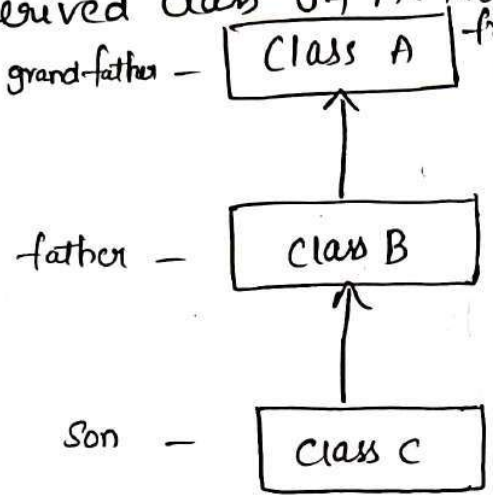
output

Total Salary: 40000.00

ii) multilevel Inheritance:

→ The Inheritance in which a class can be derived from another class is known as multilevel Inheritance.

→ Suppose there are three classes A, B and C. A is the base class that derives from class B. So, B is the derived class of A. Now, C is the class that is derived from B. Ex: //multilevel inheritance in Java Example program




```
Class A
{
}
Class B extends A
{
}
Class C extends B
{
}
```

Example:

```
import java.lang.*;
Class Add
{   int a=15, b=10;
    void add()
    {  S.o.pln(a+b);
    }
}
Class Sub extends Add
{
    void sub()
    {  S.o.pln(a-b);
    }
}
```

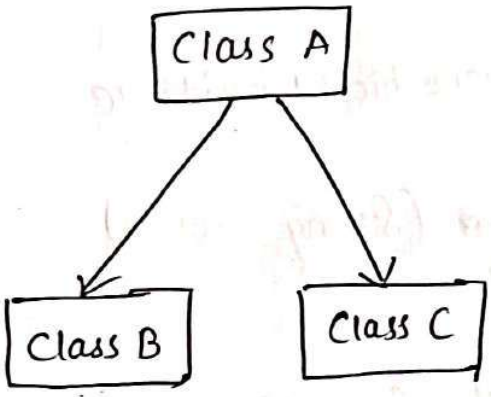
```
Class Mul extends Sub
{
    void mul()
    {
        S.o.pln (a*b);
    }
}
Class Div extends Mul
{
    void div()
    {
        S.o.pln (a/b);
    }
}
Class Inheritance
{
    psvm(String args[])
    {
        Div d = new Div();
        d.add();
        d.Sub();
        d.mul();
        d.div();
    }
}
```

Save: Inheritance.java

output: 25
5
150
1

iii) Hierarchical Inheritance:

→ Hierarchical inheritance in java is a type of inheritance where multiple child classes inherit properties & methods from a single parent class.



Example

```
import java.lang.*;
Class Animal
 {
   void eat()
    {
      S.o.pln("Eating");
    }
 }
 Class Dog extends Animal
    {
      void bark()
       { S.o.pln("Barking");
       }
    }
```

```
class cat extends Animal
{
    void meow()
    {
        S.o.pln("Meowing");
    }
}
Class HierarchicalInheritance
{
    psvm(String args[])
    {
    Cat c = new cat();
        c.meow();
        c.eat();
    Dog d = new Dog();
        d.barks();
        d.eat();
    }
}
```

Save: HierarchicalInheritance.java

Output:

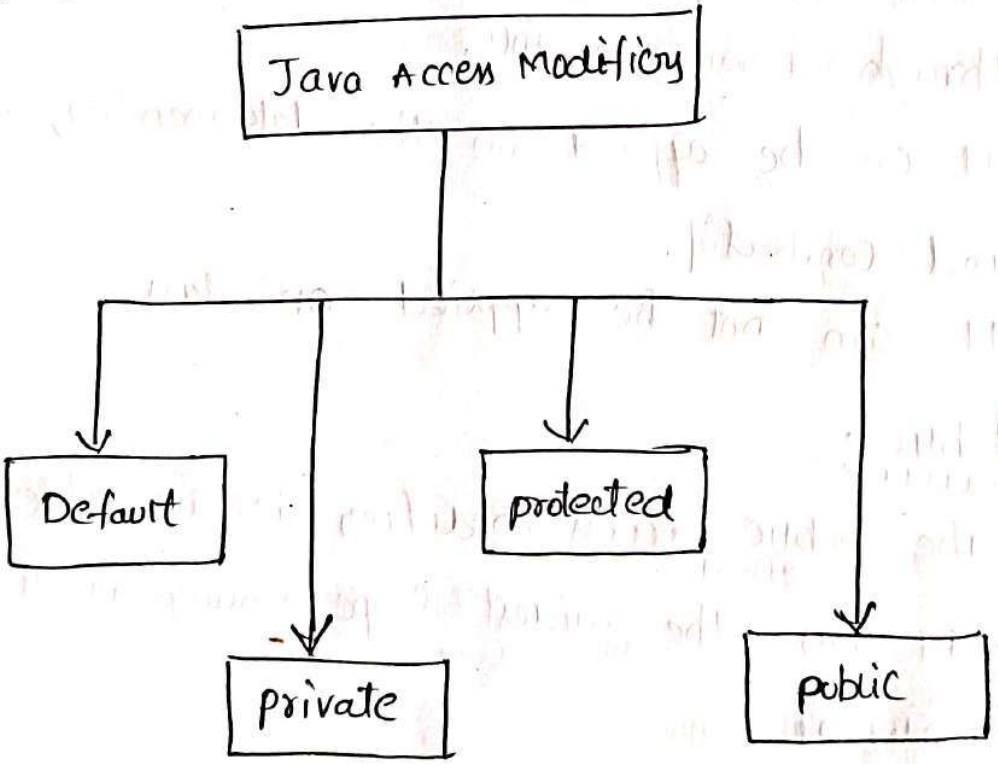
```
Meowing
Eating
Barking
Eating
```

* Access Specifiers (or) Access modifiers :

→ Java access specifiers are used to specify the scope of variables, datamembers, methods, classes and (or) constructors.

→ These help to restrict & secure the access (or, level of access) of the data.

There are four different types of access modifiers in java.



(i) Default (No keyword required)
(ii) private
(iii) protected
(iv) public

Private: (variables, methods, class) → within class access
→ outside class NOT possible

→ private access modifiers can be accessed with in the class only. we can't access the outside class.

Default: (class, method, variable)

→ If we don't use any modifier it is treated as default by default. So the default access modifier is accessible only with in the Package.

protected: (method, variable, constructor)

→ The protected access modifier can be accessible within the package & outside the package but through inheritance only.

→ It can be applied on the data member, method and constructor.

→ It can not be applied on class.

public:

→ The public access modifier is accessible every where.

→ It has the widest scope among all other modifiers

// Example program for public access modifier

*
```
package pack1;

public class One
  {
    public void show()
      {
        S.o.pln ("method in class one from pack1");
      }
  }
```

→ javac -d. One.java

*
```
package pack2;
import pack1.*;
class Two
  {
    public static void main (String args[])
      {
        One ob= new One();
            ob.show();
      }
  }
```

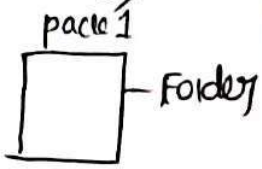
→ javac -d. Two.java
→ java pack2. Two

o/p:
method in class one from pack1.

* // Example program for protected access modifier

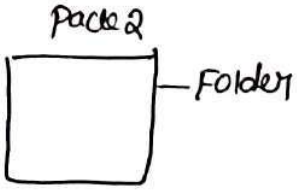
```
package pacle1;

public class A
{
    protected void show()
    {
        S.o.pln (" method in class A from pacle1");
    }
}
```



→ javac -d . A.java

*
```
package pacle2;
import pacle1.A;
class B extends A
{
    psvm (String args[])
    {
        B ob = new B();
        b.show();
    }
}
```



→ javac -d . B.java

→ java pacle2.B

O/P: method in class A from pacle1

Example

// Example program for private Access Modifier

```
import java.lang.*;

Class A
{
    private void show()
    { s.o.pln("method in class ");
    }
}

Class PrivateDemo
{
    psvm (String args[])
    {
        A ob = new A();
        ob.show();
    }
}
```

Save: PrivateDemo.java

output

Compile time Error

→ show() has private acces in A

ob.show();

1 Error

* Access Control & Inheritance :

Access Specifier in inheritance :

→ Although a Subclass includes all of the members of its Super class, it cannot access those members of the Super class that have been declared as private.

Example

```
import java.lang.*;

Class SuperTest
 {  int a; // public by default
    private int b; // private
    void setData (int x, int y)
      {  a = x;
         b = y;
      }
 }
Class SubTest extends SuperTest
  {  int tot;
     void sum()
       {  tot = a+b; // b is private not available
                     in sub class
       }
  }
```

```
Class Access
{
    psvm(String args[])
    {
        SubTest s = new SubTest();
    s.SetData(10,20);
    s.sum();
    S.o.pln("TOTal is:"+s.tot);
    }
    }
```

→ The above program give the following error:

Access.java:18: b has private access in SuperTest

```
tot = a+b; // b is private is not
                available in Sub class
```

Note:

→ variable b is declared as private. it is only accessible by other members of its own clas. Sub class have no access.

```
Class Access
{
    psvm(String args[])
    {
        SubTest s = new SubTest();
    s.SetData(10,20);
    s.sum();
    S.o.pln("Total is :" + s.tot);
    }
    }
```

→ The above program give the following error:

Access.java : 18 : b has private access in SuperTest

```
tot = a+b; // b is private is not
                available in Sub class
```

Note:

→ variable b is declared as private. it is only accessible by other members of its own class.

Sub class have no access.

Constructor Method and Inheritance:

[// write Constructor & Inheritance Defination]

→ constructor are called in the order in which they created.

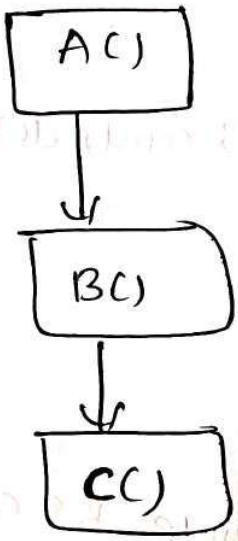
Example:

```
import java.lang.*;
Class A
{
   A()
   { S.o.pln(" Inside A'S Constructor...");
   }
}
class B extends A
   {  B()
      {S.o.pln (" Inside B'S Constructor...");
      }
   }
Class C extends B
   {  C()
      { S.o.pln("Inside C'S constructor...");
      }
   }
   Class CallingCons
      {  psvm (String args[])
         {
            C c = new C();
         }
      }
```

Note:

→ To execute Super class Constructor Super() must be the first Statement executed in a Subclass Constructor.

→ But if Super() is not used, then the default or parameter less constructor of each Superclass will be executed.



* Interface & Inheritance

Inheritance	Interface
→ Inheritance is the mechanism in java by which one class is allowed to inherit the features of another class.	→ Interface is the blueprint of the class. Like a class, an interface can have methods & variables. but the methods declared in an interface are by default abstract.
→ It is used to get the feature of another class.	→ It is used to provide total abstraction.
→ class subclass_name extends superclass_name { }	→ interface <interface_name> { }
→ It is used to provide 4 types of inheritance. (multilevel, simple, hybrid & hierarchical inheritance)	→ It is used to provide 1 types of inheritance (multiple).
→ It uses extends keyword	→ It uses 'implements keyword
→ we cannot do multiple inheritance (Causes compile-time error)	→ we can do multiple inheritance using interface

* Inheritance of Interfaces:

→ Like classes interfaces can be extended (Inherited).

→ An interface can be subinterfaced from other interfaces.

Ex①

```
interface name2 extends name1
    {
      Body of name2
    }
```

Ex②

→ we can put all constants in one interface and all the methods in another interface.

```
interface ItemConstants
{
    int code = 501;
    String name = "light";
}
interface Item extends ItemConstants
{
    void display();
}
```

→ we can also combine several interfaces into single interface

Ex:

```
interface ItemConstants
{
    int code = 501;
    String name = "Light";
}
interface Itemmethod
{
    void display();
}
interface Item extends ItemconstantsItemmethod
{
    ---;
    ---;
}
```

Note:

→ Interfaces are implemented by classes to define the methods.

→ Interfaces cannot extends classes, this will violate Rule that an interface can have only abstract methods.

* Default methods in Interfaces:

→ Methods which are defined inside the interface & tagged with default are known as default methods. These methods are non-abstract methods.

Example:

```
import java.lang.*;
interface Sayable
{
    //Default method
    default void Say()
    {
        S.o.pln(" This is default method ");
    }
    //Abstract method
    void SayMore(String msg);
}
public class DefaultMethods implements Sayable
{
    public void SayMore(String msg)
    { //implementing abstract method
        S.o.pln (msg);
    }
    psvm(String args[])
    {
DefaultMethod dm = new DefaultMethod();
```

```
dm.Say();  // Calling default method
dm.SayMore("Kalam Sir");  // Calling abstract method
}
}
```

* Application of Super Keyword :-

→ The keyword super is used by the Sub-class to refer its immediate Super class instance variables, constructors and methods.

To call Super class Constructor :

→ A Sub class can call a constructor defined by its Super class by using the following format.

```
Super(parameter list);
```

→ Super must always be the first statement executed inside a Sub-class constructor.

→ Super may only be used within a Subclass constructor

→ The parameters in the Sub-class must match the order and type of the instance variables declared in the Super-class.

Example : (Super class Keyword)

```
import java.lang.*;

class One //Super class
{
    One()
    {
      s.o.pln ("This is one class");
    }
}
Class Two extends One
   {
      Two()
      { s.o.pln ("This is Two class");
        }
    }
    Class Demo
     { psvm (String args[])
        {
          Two ob = new Two();
        }
      }
```

output

This is one class

This is Two class

```
import java.lang.*;

class One //Super class
{
    int x;
    One(int a)
    {
      x=a;
    }
}
class Two extends One //Sub-class
{     int y;
    Two(int b)
      { y=b;
      }
}
class Demo
{ psvm(String args[])
    {
     Two ob = new Two(15);
    }
}
```

output

Error

```
import java.lang.*;

class One //Super-class
{   int x;
    One(int a)
    {  x=a;
    }
}
Class Two extends One // Sub-class
{   int y;          -> Super class parameter
    Two(int a, int b)   -> sub class parameter
    {
        Super(a);
        y=b;
    }
        void display()
        {  S.o.pln(x+y);
        }
}
Class Demo
{  psvm(String args[])
    {
      Two ob = new Two(30,40);
    }
}
```

ob

x	y
10	20

void display()

* Invoking Super class Members:

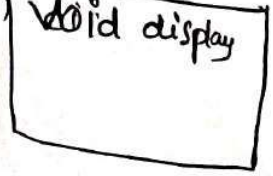
To call Super class instance variable & methods:

→ The super keyword is used when sub class members hides Super class members (Same name in Super class & sub class)

Syntax: `Super.member;`

```
import java.lang.*;
  Class One
  {   void display()
      {  S.o.pln("This is One");
      }}
  Class Two extends One
      {  void display()
          {
             Super.display();
            S.o.pln("This is Two");
          }  }
  Class Demo
      {  psvm(String args[])
          {
             Two ob = new Two();
```

ob



* Method overriding: (or) Function overriding

(i) Compiler (overloading) - method name Same but dif Sig
return type dif

(ii) Runtime (overriding) - method name Same Sign also same,
return type Same

	overloading		overriding
(i)	method name Same	———	Same
(ii)	~~method~~ Sign diff	———	Same
(iii)	diff Same return type	———	Same
(iv)	instance, static, main, constructors	———	instance
(v)	Same / child class	———	parent, child class
(vi)	Static / Early binding	———	dynamic / late Binding
(vii)	jvm compilation	———	jvm runtime

overriding:

→ Declaring a method in Sub class which is already present in Parent class is known as method overriding.

→ overriding means to override the functionality of an existing method.

→ method overriding is an example of runtime Polymorphism.

→ Static & final method cannot be overriden they are local to the class.

→ overriding ⟶ Dynamic binding ⟶ late binding

~~→ for the runtime~~

* Program 1:

```
import java.lang.*;
Class OverridingDemo // parent class
{
    void message() // Ins method
    { S.o.pln (" parent method");
    }
}
Class Demo extends OverridingDemo
{
    void message() // Ins method
    { S.o.pln ("child method");
    }
    psvm (String args[])
    {
        Demo d = new Demo();
        d.message();
    }
}
```

O/P: child method

Program 2:

```
import java.lang.*;
Class OverridingDemo
{
    void msg()
    {
        S.o.pln (" parent method ");
    }
}
Class Demo extends OverridingDemo
{
    void msg()
    {
        S.o.pln ("child method ");
    }
    psvm (String args[])
    {
        OverridingDemo d = new OverridingDemo();
        d.msg();
    }
}
```

O/p: parent method

Method Overriding (Dynamic polymorphism)

→ Overriding is a mechanism redefining the functionality of Super class method in the Sub class.

(or)

→ writing Two (or) more methods in Super and Sub classes, such that the methods have same name and same signatures is called "Method overriding. Subclass methods overrides Super class methods.

→ Dynamic polymorphism (or) late Binding

Example:

```
class Superclass
{
    void display()
    {
        S.O.pln("Hello");
    }
}
class Subclass extends Superclass
{
    void display()
    {
        S.O.pln("Bye");
    }
}
```

Example:

```
import java.lang.*;
class Superclass
 {
    void display()
     { S.o.pln ("Hai");
       }
 }
class Sub extends Superclass
 {
    void display()
      {
        S.o.pln (" Bye ");
       }
 }
class Main
 { psvm(String args[])
     {
   Sub ob =new Sub();
       ob.display();
    }
 }
```

* Interface: (Introduction)

→ An interface is a collection of Abstract methods.

→ An interface can contain both variables and methods.

→ An interface can not be instantiated.

→ In order to access the members of interface we need to inherit the interface into a class using implements keyword.

→ In the Sub class we have to override all the abstract methods.

→ If the subclass is overriding all the methods of an interface then it is called as implementation class.

→ A class can implement any number of interfaces.

Declaration Syntax:

Syntax:

```
interface interfaceName
{
    variable declaration;
    method declaration;
}
```

* Program:

```
import java.lang.*;

interface MyInterface // interface
{
   public abstract void m1(); //abstract method
}

Class InterfaceDemo implements MyInterface
{
    public void m1()
    {
      System.out.println(" Hello ");
    }
    public Static void main (String args[])
    {
  InterfaceDemo id = new InterfaceDemo();
       id.m1();
    }
}
```

output:

Hello

* Program:

```
import java.lang.*;

interface MyInterface //interface
{
public static final int x=5;
        public abstract void m1(); //abstract metho
}

class InterfaceDemo implements MyInterface
{
    public void m1()
    { S.o.pln("welcome");
    }
     psvm (String args[])
     {
InterfaceDemo id = new InterfaceDemo();
      id.m1();
      S.o.pln(id.x); (or) S.o.pln(x);
     }
}
```

output

welcome
5

* program: (Reference)

```
import java.lang.*;
interface MyInterface // interface
{
    public static final int x = 10;
    public abstract void m1(); // abstract method
}
class InterfaceDemo implements MyInterface
{
    public void m1()
    {
        S.o.pln ("welcome");
    }
    psvm (String args[])
    {
        MyInterface id = new InterfaceDemo();
        id.m1();
        S.o.pln (x);
    }
}
```

Output

welcome

10

* Program:

```
import java.lang.*;

interface MyInterface
{
  public static final int x=15;
  pubic abstract void m1();
}
interface MyInterface1
{
  pubic static final int y=20;
  pubic abstract void m2();
}
Class InterfaceDemo implements MyInterface, MyInterface1
{
    public void m1()
      {
        S.o.pln("welcome");
      }
    Public void m2()
      {
        S.o.pln("Kalam Sir");
      }
        psvm (String args[])
          {
InterfaceDemo id= new InterfaceDemo();
          id.m1();
          S.o.pln(x);
          id.m2();
          S.o.pln(y); } }
```

O/P welcome
15
Kalamsir
20

Abstraction :

→ Abstraction is a process of hiding the implementation details from the user. Only the functionality will be provided to the user.

→ In Java, abstraction is achieved using abstract classes (0-100%) and interfaces (100%).

class ----> class ----> extends (single class)

class ----> interface ----> implements (any number)

interface ----> interface ----> extends (any number)

* Program:

```
interface MyInterface // interface
{
  public static final int x=5;
  public abstract void m1(); // abstract method
}
interface MyInterface1 extends MyInterface // intrfac
{
  public static final int y=10;
  public abstract void m2(); // abstract method
}
Class InterfaceDemo implements MyInterface1
{
  public void m1()
  {
    S.o.pln ("Welcome");
  }
  public void m2()
  {
    S.o.pln ("Kalam Sir");
  }
  psvm (String args[])
  {
InterfaceDemo id = new InterfaceDemo();
id.m1();
```

```
S.o.pln (x);
id.m2();
S.o.pln (y);
    }
}
```

output

welcome
5
kalam sir
10

* program : (Multiple interfaces)

```
interface MyInterface // interface
{
public Static final int x=5;
public abstract void m1(); // abstract method
}
interface MyInterface2
{
public Static final int z=15;
public abstract void m3(); // abstract method
}
interface MyInterface1 extends MyInterface,
                                MyInterface2
{
public Static final int y=20;
public abstract void m2(); // abstract method
}
Class InterfaceDemo implements MyInterface1
{   public void m1()
    {  S.o.pln ("welcome");
    }
```

```
public void m2()
{ S.o.pln(" Kalam Sir");
}
psvm (String args[])
{
Interface Demo id = new InterfaceDemo();
id.m1();
S.o.pln(x);
id.m2();
S.o.pln(y);
id.m3();
S.o.pln(z);
}
}
```

```
public void m3()
{ S.o.pln("Java");
}
```

Output

```
welcome
5
Kalam sir
20
Java
15
```

```
import java.lang.*;
interface MyInterface
{   public static final int x=15;
    public static void m1();
}
interface MyInterface2
   {  public static final int Z=20;
      void m1();
   }
interface MyInterface1 extends MyInterface, MyInterface2
   {
public static final int y=30;
   public abstract void m1();
   }
class InterfaceDemo implements MyInterface1
   {  public void m1()
      {
         S.o.pln ("welcome");
      }
      psvm (String args[])
      {  InterfaceDemo id=new InterfaceDemo();
         id.m1();
         S.o.pln(x);
         S.o.pln(y);
```

```
        So.pln (z);
    }
}
```

Output

```
welcome
15
20
30
```

* Static method in Interface:

→ Static methods in interface are those method, which are defined in the interface with the keyword static.

→ Unlike other methods in interface, these static methods contain the complete defination of the function.

// Static Methods in Interface :- [program]

```
import java.lang.*;
interface MyInterface    // interface
{
  public static final int x=15;
  public abstract void m1(); //abstract method
}
interface MyInterface2   // interface
{
    public static final int z=25;
    public abstract void m3(); //abstract method
        static void m4()  // Static method
        { S.o.pln (" from Static method");
        }
}
interface MyInterface1 extends MyInterface, MyInterface2
{  public static final int y=20;
   public abstract void m2(); //abstract method
}
class InterfaceDemo implements MyInterface1
    {  public void m1()
        { S.o.pln("welcome");
        }
```

```
pubic void m2()
{
  S.o.pln("Kalam Sir");
}
pubic void m3()
{
 S.o.pln(" CSe-D");
}
psvm (String args[])
{
InterfaceDemo id = new InterfaceDemo();
   id.m1();
 S.o.pln(x);
   id.m2();
 S.o.pln(y);
 id.m3();
 MyInterface2.m4();
 S.o.pln(z);
}
}
```

O/P:

welcome

15

Kalam Sir

25

CSe-D

20

* Abstract class in Java:

→ A class that is declared using "abstract" keyword is known as abstract class.

→ It can have abstract methods (methods without body) as well as concrete methods (regular methods with body)

→ An abstract class can not be instantiated, which means you are not allowed to create an object of it.

→ A class derived from the abstract clay must implement all those methods that are declared as abstract in the parent class.

Ex:

```
abstract class <class-name>
{
    variable declaration;

    Void add()
    {
      ——;
      ——;
    }
    abstract void sub();
}
```

abstract method in Java:

→ abstract method has no body (only declaration no definition)

→ Always end the declaration with a Semicolon (;)

→ It must be overridden, An abstract class must be extended and in a same way abstract method must be overridden.

→ A class hay to be declared abstract to have abstract methods.

Important points:

→ Abstract class may also have concrete (complete) methods.

→ Reference of an object abstract class can point to objects of its Sub-classes there by achieving run time polymorphism.

→ A class must be compulsorily labeled abstract, if it has one (or) more abstract methods.

→ It can have Constructors & static methods also.

abstract method in Java:

→ abstract method has no body (only declaration no definition)

→ Always end the declaration with a Semicolon (;)

→ It must be overridden, An abstract class must be extended and in a same way abstract method must be overridden.

→ A class has to be declared abstract to have abstract methods.

Important points:

→ Abstract class may also have concrete (complete) methods.

→ Reference of an object abstract class can point to objects of its sub-classes there by achieving run time polymorphism.

→ A class must be compulsorily labeled abstract, if it has one (or) more abstract methods.

→ It can have constructors & static methods also.

* program:

```
import java.lang.*;

abstract class Demo1
{
    void normet()
    { S.o.pln ("normal method");
    }
    abstract void ab_method();
}
class Demo2 extends Demo1
{   void abmethod()
    {  S.o.pln ("abstract method");
    }
}
Class Mclass
{   psvm (String args[])
    {
    Demo2 ob = new Demo2();
            ob.normet();
            ob.ab_method();
    }
}
```

O/P

normal method
abstract method

* Nested Interfaces (or) inner interface (or) member interface:

→ An interface declared within another interface (or) class, is known as anested interface.

→ interface (or class) can have public and default access Specifiers when declared outside any other class.

→ interfaces contains abstract methods & static & final Variables.

Syntax:

```
interface Demo1
{
  interface Demo2
  {
    //code
  }
}
```

(or)

Syntax:

```
Class A
{
  interface Demo1
  {
    //code
  }
}
```

* Program : Filename Interface.java

```
import java.lang.*;
import java.io.*;
 interface Demo1   // interface
  {  interface Demo2 // Sub (or) nested interface
     {
          void display(); //abstract method
     }
  }
class  A implements Demo1.Demo2
   {  public void display()
        {
        S.o.pln ("nested interface method");
   }
}
   class Interface
    {  psvm(String args[])
        {
            A  ob = new A();   (or) Demo1.Demo2 A = new A();
              ob.display();
        }
    }
```

operator
object
Demo1.Demo2 → interface
A() → Constructor

Output

nested interface method

```
public void m2()
 {
   S.o.pln (" KalamSir");
 }

Psvm (String args[])
 {
InterfaceDemo ob = new InterfaceDemo();
    ob.m1();
   S.o.pln (x);
    ob.m2();
   S.o.pln(y);
    OB.m3();
   S.o.pln(z);
     }
 }
```

output

```
welcome
10
KalamSir
20
from default method
30
```

* JAVA Annotations:

→ Java Annotation is a tag that represents the metadata.

→ Annotations start with '@'.

→ Annotations do not change action of a compiled program.

→ Annotations help to associate metadata (information) to the program elements i.e. instance variables, constructors methods, classes etc.

→ Annotations are not pure comments as they can change the way a program is treated by compiler.

Java Annotation-uses:

→ Annotations have a number of uses, among them

(1) Information for the compiler:

→ Annotations can be used by the compiler to detect error (or) suppress warnings.

(2) Compile-time and deployment-time processing:

→ Software tools can process annotation information to generate code XML files.

(3) Runtime processing:

→ Some annotations are available to be examined at runtime.

Example:

→ The annotation's name is override:

```
@Override
void mySuperMethod()
{
  ___;
  ___;
  ___;
}
```

→ The annotation can include elements, which can be named (or) unnamed, and there are values for those elements:

```
@Author (name = "Ramanujan", date = "4/29/2001")
Class MyClass()
{
  ___;
  ___;
  ___;
}
```

Example:

```
import java.lang.*;
Class Animal
{
  void eatSomething()
  { S.o.pln("eating Something");
    }
  Class Dog extends Animal
    {
      @override
      void eatsomething()
       {S.o.pln("eating foods");
        }
      Class TestAnnotation
       {
      public Static void main(String args[])
        {
          Animal a = new Dog();
          a.eatSomething();
        }}
```

→ javac TestAnnotation.java

TestAnnotation.java 8 : error : method does not override (or) implement a method from a Supertype

```
@override
^
```

1 error

* Functional Interface:

→ An interface that contains exactly one abstract method is known as functional interface. It can have any number of default, Static methods but can contain Only One abstract method.

→ Functional Interface is also known as Single Abstract Method Interfaces (or) SAM Interfaces. It is a new feature in java, which helps to achieve functional programming approach.

Example

```
import java.lang.*;
interface Demo  // interface
{  void display(String msg); //abstract method
}
public class FunctionalInterfaceDemo implements Demo
{
    public void display(String msg)
    {
        System.out.println(msg);
    }
    public Static void main(String args[])
    {
FundionalInterfaceDemo fid = new FunctionalInterfaceDemo();
        fid.display("Kalam Sir"); } }
```

output:

Kalam Sir

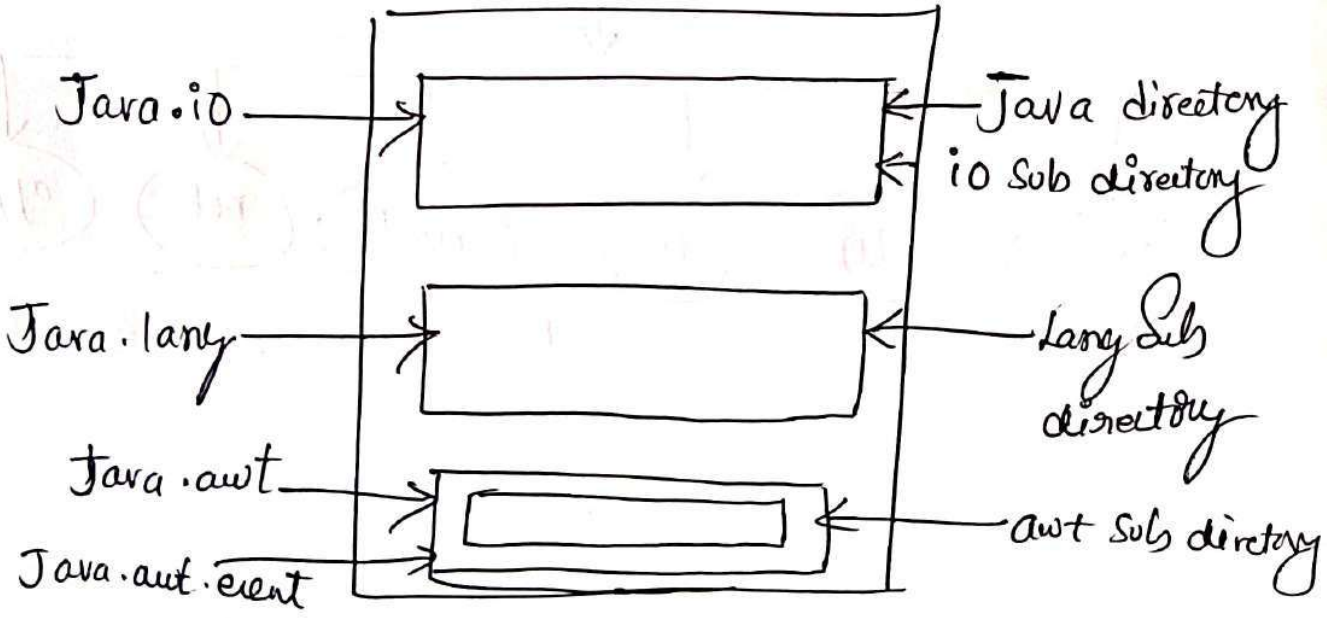
* Packages and Java Library : (Introduction)

→ It is necessary in software development to create several classes & interfaces.

→ After creating these classes & interfaces, it is better if they are divided into some groups depending on their relationship.

→ So these classes & interfaces are stored in some directory.

→ The directory (or) folder is known as Package.



Advantages

→ Two classes in two different package can have the same name

→ A group of packages is called a library. The reusability nature of packages makes programming easy.

There are two types of packages

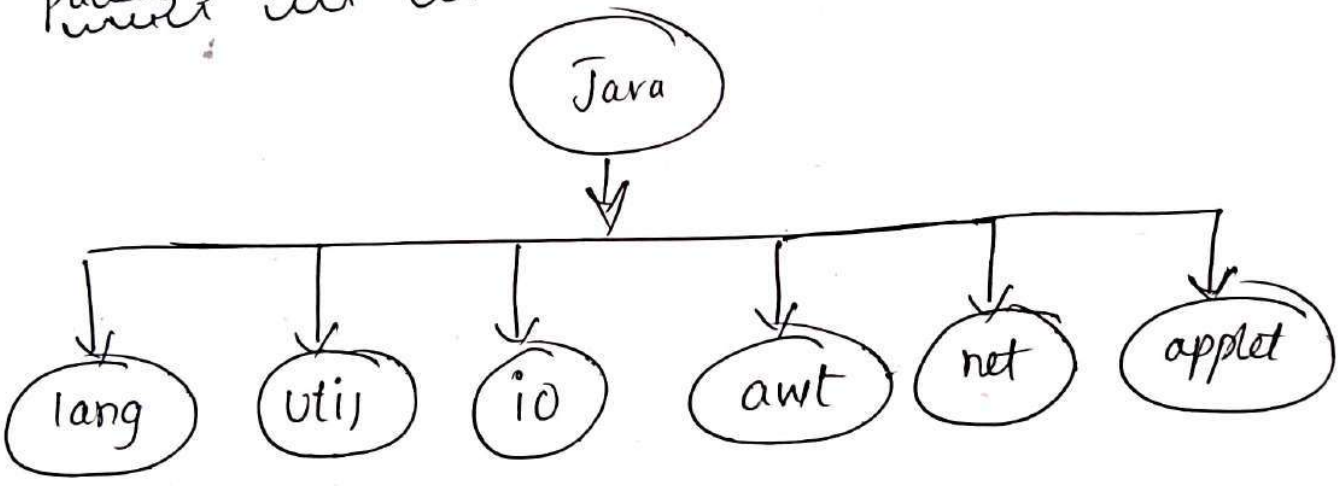
(1) Built-in package / Java API package

(2) User Defined packages

Java API packages:

→ Java API provides a large number of classes grouped into different packages according functionality.

packages from java API are:



* Defining Package:

→ A package is created using keyword package.

Syntax

package <package>;

→ package statement must be first statement in a java source file.

fx:

```
package Studentpack; // package declaration

Class Student // class declaration
{
 ---;
 ---;
 ---;
}
```

Create a package which Contains Addition class in that:

```
package pacle;

public class Addition
  {
     private double a,b;
     public Addition(double a, double b)
       {
         this.a;
         this.b;
       }
     public void Sum()
     {
       S.o.pln("sum"+(a+b));
     }
  }
```

Compiling the Program:

```
>javac -d. Addition.java
```

→ -d tells java compiler to create a Separate Sub directory & place the .class file there.

→ Dot(.) indicates that the package should be created in the current directory.

Program to use Addition.class from mypacle:

```
class Usepacle
{ psvm (String args[])
  {
  pacle.Addition obj = new pacle.Addition (10,30.5);
   obj.sum();
  } }
```

Note

→ Instead of referring the package name every time, we can import the package like this.

```
import pacle.Addition;
```

→ then the program can be written as

```
import pacle.Addition;
class UsePacle
{  psvm (String args[])
    {  Addition obj = new Addition(10,30.5)
       obj.sum();
    } }
```

* importing package and classes into programs:

→ There are two ways of accessing classes stored in a package.

(1) Fully Qualified classname

(2) Using Import Statement.

(1) Fully Qualified class name

```
double y = java.lang.Math.Sqrt(x);
```

(2) Using Import Statement:

Syntax:

```
import packagename.classname;
```

(or)

```
import packagename.*;
```

```
import java.awt.Font;
```

— import Font class in our program

(or)

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

— import all classes of awt package in our program.

path and Class Path :

PATH	CLASSPATH
→ PATH is an environment variable.	→ CLASSPATH is an environment variable that tells the java compiler where to look for class files to import. Generally CLASSPATH is set to a directory.
→ It is used by the operating System to find the executable files (.exe)	→ It is used by the Application class loader to locate the .class file.
→ you are reavired to include the directory which contains .exe files.	→ you are reavired to include all the directories which contain .class and JAR files.
→ PATH enivornment variable once set, cannot be overridden.	→ The CLASSPATH enivornment variable can be overridden by using the command eine option -cp.

How to Set CLASSPATH in windows using Command Prompt:

→ CLASSPATH is an environment variable that tells the java compiler where to look for class files to import. Generally CLASSPATH is set to directory (or) JAR file

→ Type the following command in your command prompt and press enter.

```
set CLASSPATH = %CLASSPATH;
C:\Program File\Java\jre1.8\rt.jar;
```

In the above command, the set is an internal DOS command that allows the user to change the variable value.

- CLASSPATH is a variable name. The variable enclosed in percentage sign (%) is an existing environment variable.

* Access Control:

Access Location \ Access Modifier →	private	public	Default	proteded	private protected
Same package Same class	yes	yes	yes	yes	yes
Same package Same class	NO	yes	yes	yes	yes
Same package NON-Subclass	NO	yes	yes	yes	NO
Different package Subclass	NO	yes	NO	yes	yes
Different package NON-Subclass	NO	yes	NO	NO	NO

→ This table only applies to member of classes.

→ A class has only two access levels.

Public – accessible any where

Default – accessible with in the Same Package.

* packages in Java SE:

Java Standard Edition provides Packages namely -

→ applet - This package provides classes & methods to create & communicate with the applets.

→ awt - This package provides classes & method to create user interfaces.

→ io - This package contains the classes & methods to read & write data standard input and output devices and files.

→ lang - This package contains classes, & methods which helps you to and interfaces of Java language.

→ net - This package ~~contains~~ provides classes to implement networking applications.

→ Security - This package provides classes and interfaces for Security framework.

→ text - This package provides classes and interfaces to handle text.

→ time - This package provides API for dates, times and durations.

* Exception Handling:

→ Exception is a runtime error.

(or)

Any abnormal event in a program is called an Exception

→ Exception Handling is a mechanism it process the generated exception. (It can handle runtime error)

→ Through Exception Handling we can ensure the program is not terminated abnormally and users data not lost.

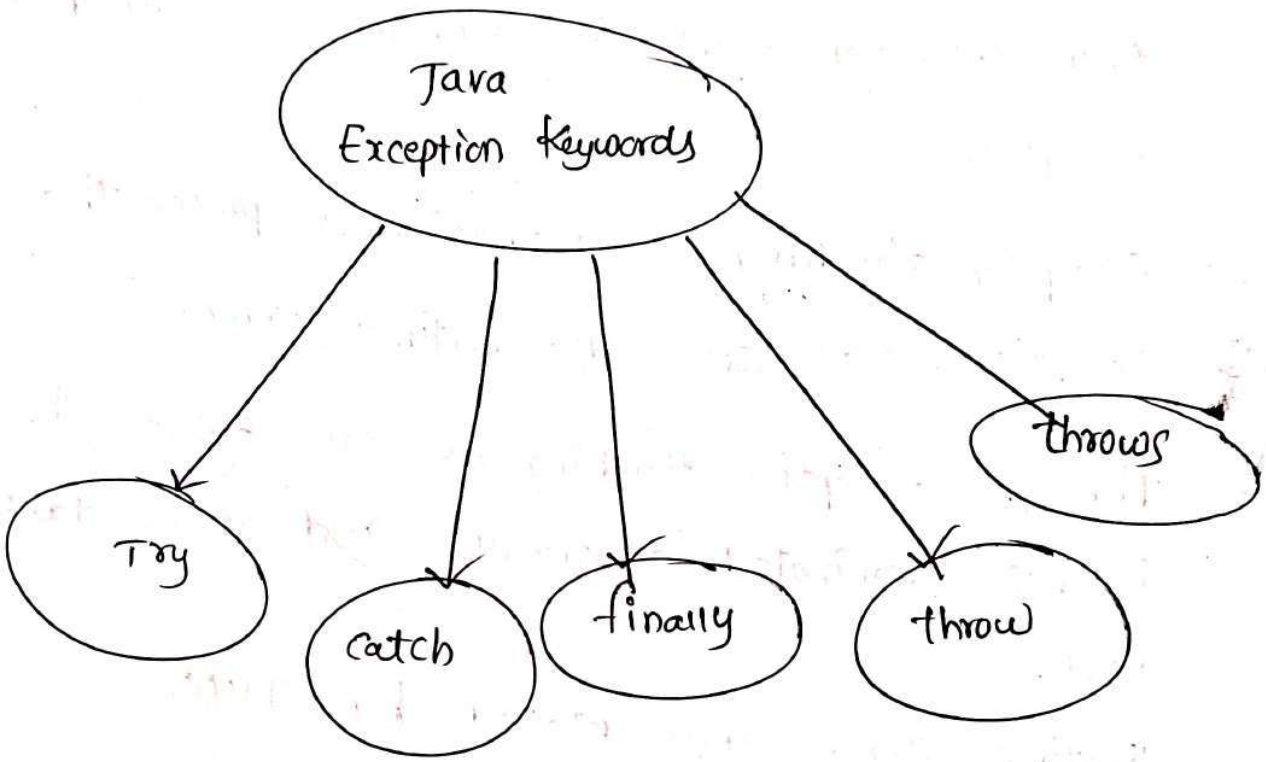
→ Runtime errors are detected by JVM.

Note

→ In Java without Exception Handling '0' (Zero) will generated the Exception and then program will be terminated. No user understandable message will be displayed.

Java Exception Keywords

→ Java provides five keywords that are used to handle the exception.



(1) Try :

→ It is a keyword used to create block statements, which are doubtfull of generating Exception.

Syntax :

```
try
{
Set of Statements → ---;
                    ---;
                    ---;   doubtfull
}
```

(2) Catch :

→ It is a keyword which is used to handle the Exception.

→ It contains blocks of Statements (Handling Code).

Syntax:

```
Catch(Type of Exception ExceptionObject)
{
Set of Statements { ——; ——; ——; } (Handling Code)
}
```

Example

```
try
{
  ——;
  ——;
  ——;
}
catch(Type of Exception Exception Object)
{
  ——; //Handling Code
  ——;
}
```

Exception

(1) a=10 b=20
→ c = b/a = 20/10 = 2

(2) a=10 b=0
→ c = b/a = 0/10 = 0

(3) a=0 b=10
→ c = b/a = 10/0 = ∞ (Infinity)

Runtime Exception Divide by zero

Exception

Ex

```
int x[] = {11,12,13};
S.o.pln(x[5]);
```

Runtime error

Array Index out of Bound Exception

* Program :(Divide by Zero Exception)

```
import java.lang.*;

Class DivideByZeroDemo
{
  psvm(String args[])
   {
     int x=10, y=0, z;
    try
    {
      z=x/y; // Divide by zero Exception
      S.o.pln(z);
     }
    catch(ArithmeticException e)
    {
      S.o.pln("DIVISION BY ZERO EXCEPTION");
     }
   S.o.pln(" program completed");
    }
}
```

output

DIVISION BY ZERO EXCEPTION

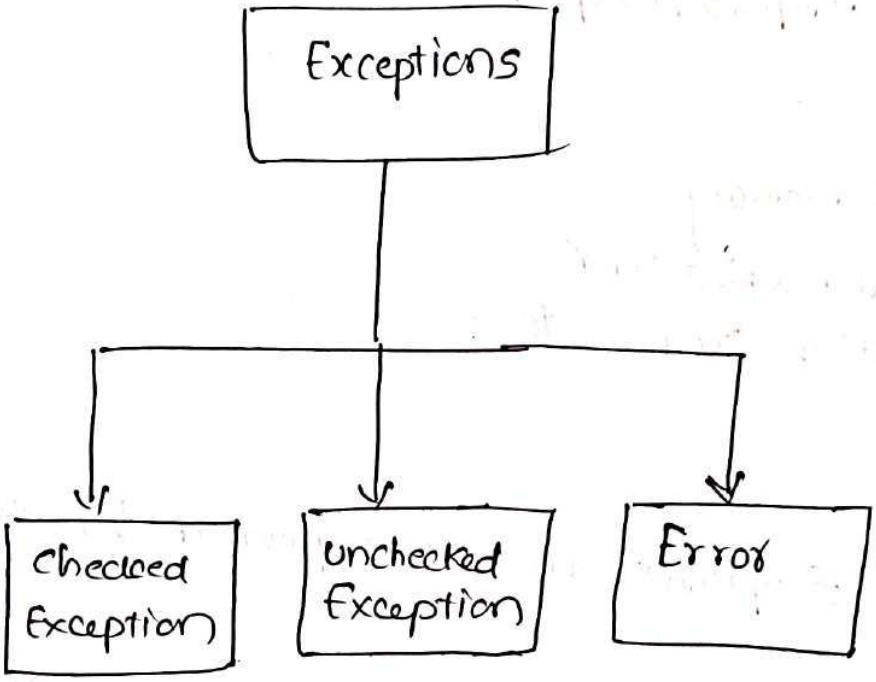
program completed

* Types of Exceptions

Exception:

→ An Exception is a condition that is caused by a runtime Error.

There are 3 types of Exceptions



(1) Checked Exception:

→ The Exception which are detected at compile time.

→ The classes that extend Throwable class except RuntimeException and Error are known as checked Exception.

Example

IoException, SqlException, FileNotFoundException).... etc

(2) Unchecked Exception:

→ The Exception which are detected at runtime

→ The classes that extends RuntimeException are known as unchecked Exception.

Example

→ ArithmeticException

→ NullPointerException ...etc

(3) Error:

→ out of memory

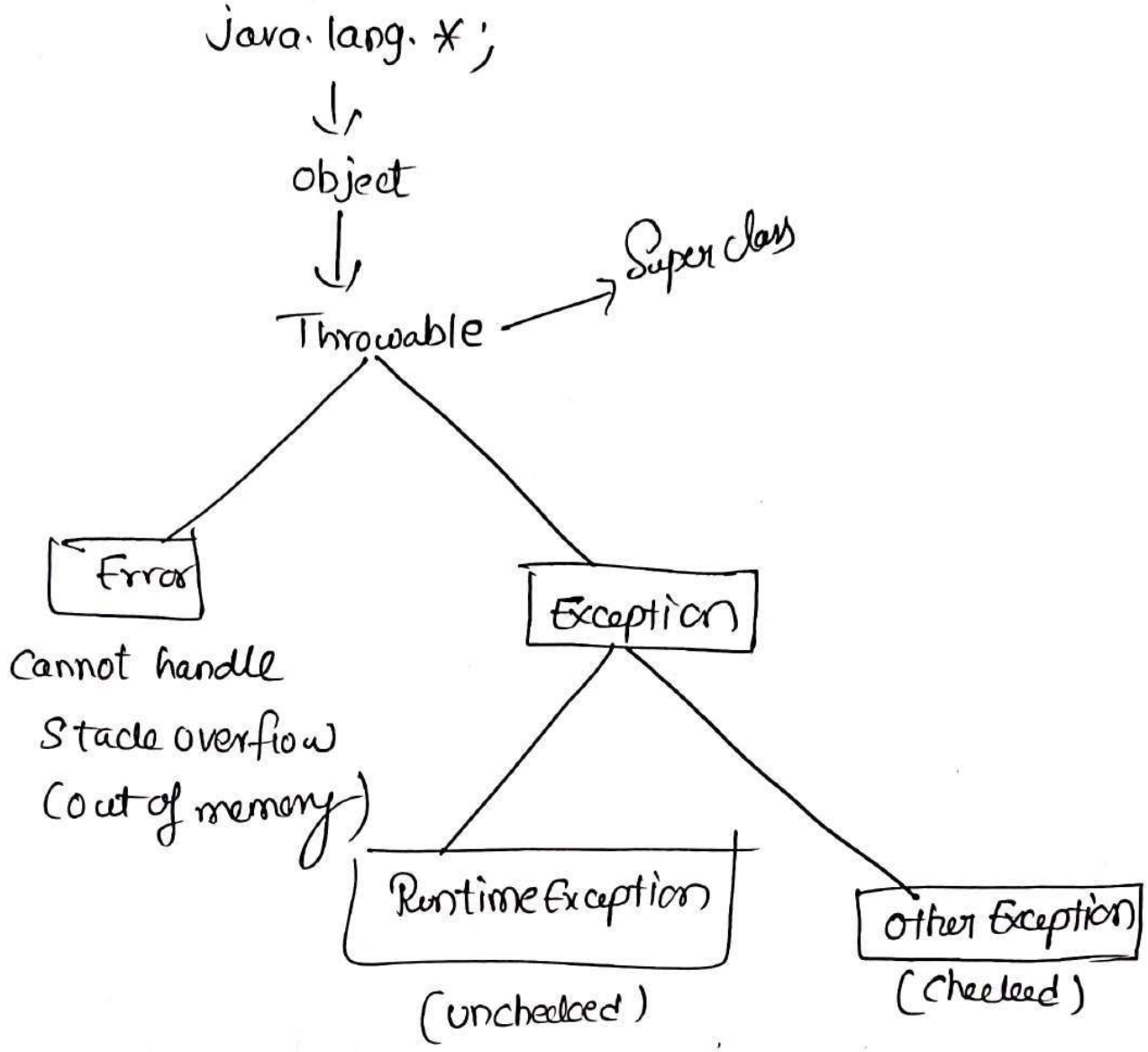
→ virtual machine error

→ Assertion error --etc

Note:

→ Every Exceptions are also represented by classes in Java.

→ All Exceptions are Subclasses of exception class


java.lang.*;
object
Throwable
Super class
Error
Cannot handle
Stack overflow
(out of memory)
Exception
RuntimeException
(unchecked)
Other Exception
(checked)

* Checked and unchecked Exception:

→ There are two types of Exceptions

(i) Checked Exceptions
(ii) Unchecked Exceptions

(1) Checked Exceptions:

→ The Exception classes that are derived from java.lang.Exception class are called checked-Exceptions.

→ All checked Exceptions must be handled by programmer explicitly otherwise compile time error occurs.

→ The java compiler checks try & catch block (or) throws clause for this kind of Exception.

→ All application Specific exceptions are come under this category.

→ Checked Exceptions do not include RuntimeException Class and its Subclasses.

Unchecked Exceptions:

→ The Exception classes that are derived from java.lang.RuntimeException class are called unchecked Exception.

→ All unchecked Exceptions are handled by System-implicitly.

→ Handling unchecked are optional by programmer.

→ unchecked Exceptions are handled by programmer explicitly to display user friendly error messages only

→ The Java compiler does not check try and catch block (or) throws clause for this kind of Exceptions.

→ All general Exceptions are come under this category.

* User Defined Exceptions in Java (or) custom Exceptions

→ In Java, you can create your own exceptions by extending the Exception class.

→ User Defined Exceptions are useful when you want to handle specific error conditions in your application that are not covered by the Standard java exceptions.

Throwable
↓
Exception
↓ extend
CustomException

(i) Define a new class that extends the Exception class.

(ii) Create a constructor that accepts a String parameter which is the Error message. This constructor will call the Constructor of the Superclass Exception.

```
Class MyCustomException extends Exception
{ public MyCustomException(String message)
  {
    Super(message);
  }
}
```

1) Exception class
2) object create
3) throw object
4) catch handle

* Example program

```
import java.lang.*;
import java.util.Scanner;

class customException extends Exception
{
  pubic customException (String message) -1
    { super(message);
    }
}

public class UserDefinedExceptionMain
{ pubic static void main(String[] args)
  {
    try
    {
    int a;
    Scanner sc = new Scanner(System.in);
  S.o.pln ("Enter a number:");
  a = sc.nextInt();
   if(a<0)
   {
throw new customException("Number cannot be negtive");
 } else
  { S.o.pln("Number is:"+a);
   } }
```

```
catch (CustomException e)
{
  S.o.pln("Custom Exception :" + e.getMessage());
}
}
```

Output:

Enter a number : -2

Number cannot be Negative

* Throwable :

→ java.lang.Throwable is a Super class for all types of Errors and exceptions in java.

→ Only instances of this class (or) its subclass are thrown by the java virtual machine (or) by the throw statement.

→ The only argument of catch block must be of this type (or) it's sub classes.

→ if you want to create your own customized exceptions, then your class must extend this class (or) its sub classes.

Example:

→ write a program of UserDefined Exception

* Built-in Exceptions in java:

→ which are available as part of java.lang package.

(1) Arithmetic Exception:

→ Runtime Error during Arithmetic operations such as Division by zero.

(2) ArrayIndex out of Bound Exception:

→ Refering to the position of the element that is not in an array.

(3) StringIndex out of Bound Exception:

→ Refering character position of the element that is not in array.

(4) Null Pointer Exception

→ Referring to the method which contains empty Object (NULL)

(5) File Not Found Exception:

→ File not found in the directory.

(6) Io Exception:

→ General failures such as in ability to read from a file.

(7) NumberformatException:

→ passing one format instead of another where a conversion between String and number.

(8) InterruptedException:

→ one processor interrupt another processor.

(9) NoSuchMethodException:

→ calling a method which is not available.

(10) ArrayStorageException:

→ These are the exceptions created by the user.

* Throw keyword:

→ Throw keyword used to create an exception and throw it explicitly.

→ throw our own exception using throw keyword.

→ Flow of execution stops immediately after the throw statements and any subsequence statements are not executed.

Syntax:

```
throw new ExceptionType("Content");
          ↳ Any type Exception
```

keyword operator

Example:

```
import java.lang.*;
Class ThrowDemo
{
  psvm(String args[])
  {
    try
    {
      throw new ArithmaticException("This is my own Exception");
      // s.o.pln("This is try block");
    }
    catch(ArithmaticException e)
      ↳ class
    { s.o.ph(e.getMessage());
    }
```

```
        S.o.pln (" program over ");
    }
}
```

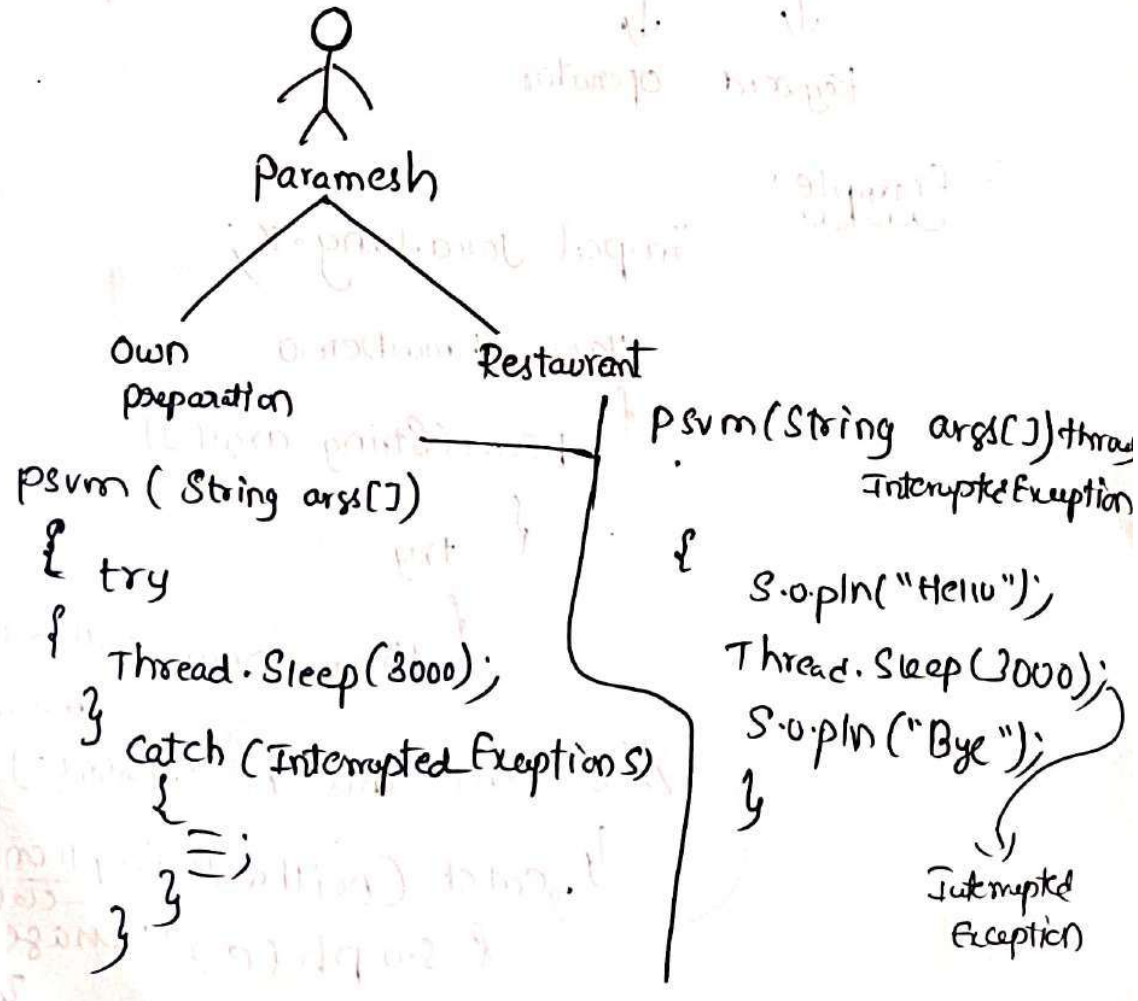
Output:

This is my own exception
program over

* Throws keyword:

→ It is a keyword, it contains list the type of exceptions that a method may throw.

→ Throws keyword is used when we doesn't want to handle the exception and try to send the exception to the JVM



Syntax

returntype methodname (parameters) throws exceptionList

```
{
    ——;
    ——;
    ——;
}
```

* Program:

```
import java.lang.*;
class Demo
{
static void display() throws ArithmaticException
  {
    int x = 10;
    int y = 0;
   int result = x/y;
    S.o.pln (result);
   }
     psvm (String args[])
      {
        try
        {
          display();
        }
        catch (ArithmaticException e)
        { S.o.pln("Division by zero");
        }
      }
}
```

O/P

Division by zero

* Finally Block:

→ It is a keyword used to create a block of code that will be executed after a try/catch block whether exception is handle (or) not.

→ If an exception occurs then catch block executes after that finally block will be executed.

→ if no exception, then finally block executed.

→ In case there is an exception, but there is no matching catch block, then the code execution skips the rest of code and executes the finally block.

→ A Finally block appears at the end of the catch block.

→ The Finally block follows a try block (or) a catch block.

Syntax

```
try
{
  //doubtful code
}
finally
{
  Executable code
}
```

2) try
{
//doubt-full code
}
catch(Ex)
{ = //handling code
}
finally
{ =
=
}

* program:

```
import java.lang.*;
class Demo
{
  psvm(String args[])
  {
    try {
          int x,y,result;
          x=20;
          y=0;
          result = x/y; //Division by zero
          S.O.pln(result);
        }
    Catch(Arithmatic Exception e)
        {
          S.O.pln("Division by zero");
        }
```

```
finally
 {  S.o.pln ("This is finally blocde");
    }
  S.o.pln ("program over");
     }
   }
```

Output

Division by zero

This is finally blocde

program over

* Multiple Catch Statements :

→ It is possible to have more than one catch statement in the catch block.

Syntax :

```
try
 {
   statements; //generates an exception
 }
 catch(Exception-type-1 e)
  {
    statements; //handles the exception
  }
 catch(Exception-type-2 e)
  {
   statements; // handles the exception
   }
   ....
   ....
   catch(Exception-type-n e)
    {
      statements; //handles the exception
    }
```

* Example program for multiple statements

```
import java.lang.*;
Class MultipleCatchDemo
{
  psvm(String args[])
   {
   try
  {
   S.o.pln(10/0);
   }
  Catch(NullpointerException e)
  { S.o.pln(e);
   }
  Catch(ArithmeticException e)
   {
     S.o.pln(e);
   }
  Catch(NumberFormatException e)
   {
     S.o.pln(e);
    }
   }
  }
```

O/P:

java.lang.ArithmeticException: / by zero

* Rethrowing Exception:

```
Class ThrowExcep
 {
    Static void fun()
     { try
      {
      throw new NullpointerException ("demo");
      }
     Catch (NullPointerException e)
      {
       S.o.pln("Caught inside fun()");
       throw e; // rethrowing the exception
      }
     }
   psvm (String args[])
    {
      try
      { fun();
      }
      Catch (NullPointerException e)
        { S.o.pln ("caught in main");
        }
      }
      }
```

O/P:

```
Caught inside fun()
Caught in main
```

An exception may be rethrown

> when an exception is rethrown and handled by the catch block.

↳ The compiler verifies that the type of re-thrown exception is meeting the conditions that the try block is able to throw.

↳ and there no other preceeding catch blocks that can handle it.

* Random class :

→ Random class is a part of java.util package. Random class is used to generate random numbers.

→ This class provides several methods to generate random numbers of type integer, double, long, float etc.

→ In Java, there is multiple ways to generate random numbers using the method and classes.

- ↳ using random() method
- ↳ using Random class
- ↳ using the ThreadLocalRandom class

* Program

```
import java.util.Random;

public class JavaTestClass
{
    psvm(String args[])
    {
        int number;
    Random random = new Random();
        number = random.nextInt(100);
    S.O.pln("Generated number is" + number);
    }
}
```

output :

30

* Enumeration (Enum) [Enum is a special class, represents a group of constants]

→ The Enum in java is a datatype which contains a fixed set of constants.

→ It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY), directions (NORTH, SOUTH, EAST, and WEST), COLORS (RED, BLUE, GREEN, WHITE and BLACK) etc.

According to the Java naming conventions, we should have all constants in capital letter. So, we have enum constants in capital letters.

→ Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly.

→ Enums are used to create our own datatype like classes. The enum datatype (also known as Enumerated Datatype) is used to define an enum in Java.

→ Unlike C/C++, enum in Java is more powerful. Here, we can define an enum either inside the class (or) outside the class.

Example: (outside a class)

```
import java.lang.*;

enum Level
{
    LOW,
    MEDIUM,
    HIGH
}

class EnumDemo
{
  psvm(String args[])
    {
    Level myvar = Level.MEDIUM;
     S.o.pln(myvar);
    }
}
```

O/p

MEDIUM

*Example: (Inside a class).

```
Class EnumDemo
{
    enum Level
    {
        LOW,
        MEDIUM,
        HIGH
    }
    psvm (String args[])
    {
        Level myvar = Level.MEDIUM;
        S.o.pln (myvar);
    }
}
```

O/P

MEDIUM

* Example (Inside a class).

```
Class EnumDemo
{
    enum Level
    {
        Low,
        MEDIUM,
        HIGH
    }
    psvm (String args[])
    {
        for (Level z : Level.values())
        {
            S.o.pln(z);
        }
    }
}
```

o/p

```
LOW
MEDIUM
HIGH
```

* Formatter class:

→ Formatter class outputs the formatted output. It can format numbers, Strings and time and date. It operates in a manner similar to the c/c++ printf function.

→ The java.util.Formatter class provides support for layout justification and alignment, common format for numeric, String and date/time data, and locale-specific output.

java.util

Class Formatter

java.lang.object
  java.util.Formatter

All implemented interfaces:
Closeable, Flushable, AutoCloseable

* Program:

```
public class FormatterDemo
{
    psvm (String args[])
    { Formatter formatter = new Formatter();
/* In this Example, the format Specifiers, %s and %d
* are replaced with the arguments that follow the format String
* %s is replaced by "Kalam", %d is replaced by 60
* %s specifies a String, and %d Specifies an integer value. All
* other characters are simply used as-is.
*/
formatter.format("%s age is %d", "Kalam", 60);
S.o.pln(formatter.out());
formatter.close();
    }
}
```

output

Kalam age is 60

* Wrapper classes in Java:

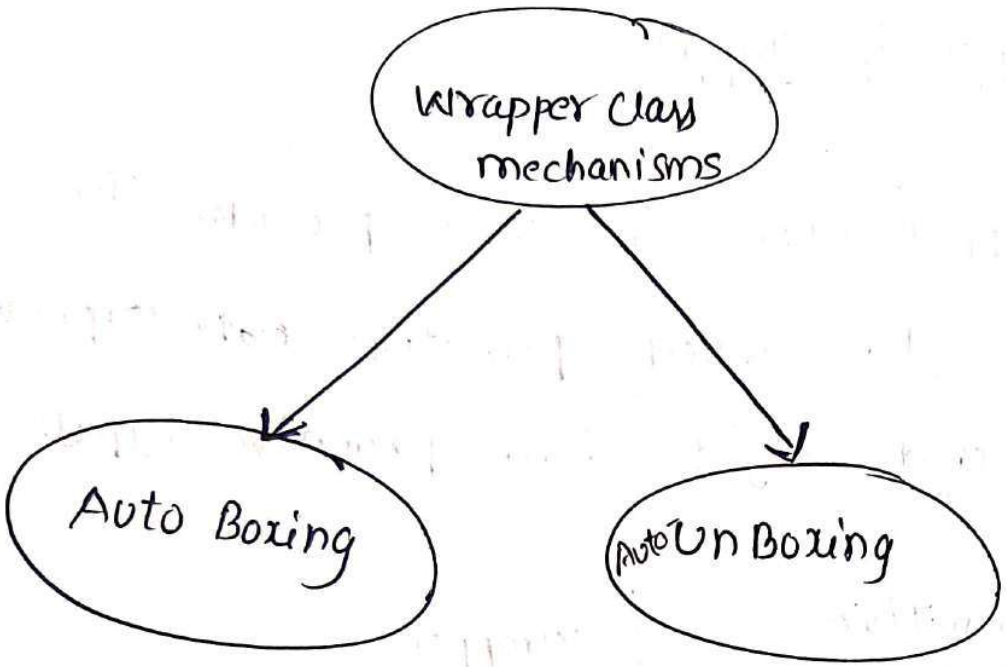
→ The wrapper class in java provides the mechanism to convert primitive data types into objects and objects into primitive types.

primitive Data Types	Wrapper classes
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

To implement wrapper classes 2 mechanisms are used.

(i) Auto Boxing

(ii) un Boxing


Wrapper Class
mechanisms
Auto Boxing
Auto UnBoxing

(1) AutoBoxing:

→ The Automatic conversion of primitive Datatype to object is called "AutoBoxing".

Example:

```
// Convert primitive DT into object
    import java.lang.*;

    Class WrapperClass
    {
      psvm (String args[])
      {
        int a=200;

      Integer ob1 = new Integer(a);
       // Boxing process with constructor

      Integer ob2 = Integer.valueof(a);
       // using valueof() method

        Integer ob3 = a; // AutoBoxing
          S.o.pln (ob1 +" "+ ob2+" "+ob3);
        }
      }
```

ob1 | 200

ob2 | 200

ob3 | 200

output

200 200 200

Auto
(2) UnBoxing :

→ The Automatic conversion of object to primitive type is called "UnBoxing".

Example :

```
// Convert object type to primitive type

import java.lang.*;

Class Wrapperclass
{
  psvm (String args[])
    {
      int a = 100;
Integer ob = new Integer(a);
//using int value() method
  int x = ob.intvalue();
  int y = ob;
S.o.pln(a+" "+x+" "+y);
}
}
```

ob 100

x 100

y 100

output

100 100 100

* Math class :

→ Math class in java is a part of the java.lang package, which means it is automatically available to all java programs.

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

The java Math class has many predefined methods that allow you to perform mathematical tasks on numbers. Here are some examples of these methods.

(1) Math.max(x,y) : Returns the number with the highest value of x and y.

(2) Math.min(x,y) : Returns the number with the lowest value of x and y.

(3) Math.sqrt(x) : Return the square root of x.

(4) Math.abs(x) : Return the absolute (positive) value of x.

(5) Math.random() : Return a random number between 0.0 (inclusive) and 1.0 (exclusive)

(6) Math.abs() : abs() method takes one parameter that is of number type and returns positive value of the number, without

using the negative sign.

For Example

The absolute value of -7 is 7.

(v)

```
import java.lang.*;
class MathclassDemo
{
    psvm(String args[])
    {
        S.o.pln (Math.random());
    }
}
```

0 to 1

Output

0.12457864211

(vi)

```
import java.lang.*;
class MathclassDemo
{
    psvm (String args[])
    {
        S.o.pln (Math.ceil(5.7));
        S.o.pln (Math.ceil(5.1));
        S.o.pln (Math.floor(5.1));
        S.o.pln (Math.floor(5.7));
    }
}
```

Output:

6.0
6.0
5.0
5.0

(vii)

```
import java.lang.*;
class MathClassDemo
{
  psvm(String args[])
    {
      S.o.pln(Math.pow(2,5));
    }S.o.pln(Math.pow(4,3));
}
```

o/p:

32.0

64.0

25

$\Rightarrow 2 \times 2 \times 2 \times 2 \times 2$

$= 32$

43

$\Rightarrow 4 \times 4 \times 4$

$\Rightarrow 64$

(iii)

Class ~~Math.SqrtClass~~

```
import java.lang.*;
Class MathClassDemo
 {
    psvm (String args[])
     {
       S.o.pln (Math.Sqrt(64));
     }
 }
```

Output

8.0

(iv)

```
import java.lang.*;
Class MathClassDemo
 {
    psvm (String args[])
     {
       S.o.pln (Math.abs(-7.5));
     }
 }
```

Output

7.5

*

(i)

```
import java.lang.*;

Class MathClassDemo
{
  psvm (String args[])
   {
    System.out.println(Math.max(4, 8));
   }
}
```

Output

8

(ii)

```
import java.lang.*;

Class MathClassDemo
{
  psvm (String args[])
   {
    S.o.pln ('Math.min(4,8));
   }
}
```

Output:

4

* Java util classes and Interfaces:

Classes of Util package :

- → ArrayDeque
- → ArrayList
- → Array
- → Bitset
- → Calendar
- → Collections
- → Currency
- → Date
- → Dictionary
- → EnumMap
- → EnumSet
- → Formatter
- → HashMap
- → Hashtable
- → HashSet
- → IdentityHashMap
- → LinkedHashMap
- → LinkedHashSet
- → LinkedList
- → Locale
- → PriorityQueue
- → Property
- → PropertyPermission
- → Random
- → Scanner
- → ServiceLoader
- → SimpleTimeZone
- → Stack
- → TimerTask
- → StringTokenizer
- → TreeMap
- → TimeZone
- → Treeset
- → Vector
- → WeakHashMap

Interfaces :-

- → Collection<E>
  - → RandomAccess
  - → Set<E>
- → Comparator<T>
- → Enumeration<E>
  - → SortedMap<K,V>
  - → SortedSet<E>
- → Deque<E>
- → Iterator<E>
- → List<E>
- → Map<K,V>
- → Map.Entry<K,V>
- → Observer
- → Queue<E>

* Java.lang package and its classes:

→ The most important classes are of lang are

object, which is the root of the class hierarchy, and class, instances of which represent classes at runtime.

→ protected object clone()
→ boolean equals (Object obj)
→ protected void finalize()
→ class getClass()
→ int hashCode
→ void notify()
→ void notifyAll()
→ void wait()
→ String toString()

→ The Wrapper classes

↳ Boolean
↳ Character
↳ Integer
↳ Short
↳ Byte
↳ Long
↳ Float
↳ Double

→ The classes String, StringBuffer and StringBuilder similarly provide commonly used operations on character strings.

Java time package - Formatter class

* Formatting for Date/Time in Java:

→ The java.time, java.util, java.sql and java.text Packages contains classes for representing date and time.

* import java.text.SimpleDateFormat;

```
import java.util.Date;

Class SimpleDateFormatExample
{
    psvm (String args[])
    {
        Date date = new Date();
SimpleDateFormat formatter = new SimpleDateFormat("MM/dd/yyyy");
String strDate = formatter.format(date);
S.o.pln("Date Format with MM/dd/yyyy:"+strDate);
formatter = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");
StrDate = formatter.format(date);
S.o.pln("Date Format with dd-M-yyyy hh:mm:ss:"+strDate);
}
}
```

Output:

Date Format with MM/dd/yyyy : 05/18/2024
Date Format with dd-M-yyyy hh:mm:ss: 18-5-2024

* Instant class - Java Time package:

→ In Java language, the Instant class is used to represent the specific time instant on the current timeline.

→ The Instant class extends the object class and implements comparable interface.

Example

```
import java.time.Instant;
public class TimeInstantDemo
{ psvm(String args[])
  {
   Instant now = Instant.now();
  S.o.pln ("Now Time = " + now);
Instant before = Instant.now().minusSeconds(600);
S.o.pln ("before Time =" + before);
Instant later = Instant.now().plusSeconds(900);
S.o.pln ("later Time = " + later);
  } }
```

o/p

```
Now Time    = 2024-06-12T02:25:19.772322600Z
before Time = 2024-06-12T02:15:19.800143002
later Time  = 2024-06-12T02:40:19.804155800Z
```

* Temporal Adjuster

→ Temporal Adjusters is used to perform the date mathematics.

Example

→ get the "next Friday of the month" (or) first day of month (or) first day of year (or) "Next Monday" etc..

* Program:

```
import java.time.LocalDate;

public class TemporalAdjustersTest
{
LocalDate todayDate = LocalDate.now();
    S.o.pln(todayDate);
  }
}
```

Output

2024-10-23

* program

```
import java.time.LocalDate;

public class TemporalAdjustersTest
{
    psvm(String args[])
    {
LocalDate todaysDate = LocalDate.now();
    S.O.pln(todaysDate);

LocalDate firstDayofMonth = todayspate.with(Temporal
                            Adjusters.firstDayofMonth());
    S.O.pln(firstDayofMonth);
LocalDate nextFriday = todayDate.with(Tempora1 Adjusters.
                       next(Dayofweek.FRIDAY));
    S.O.pln(nextFriday);
    }
}
```

output

```
2021-05-28
2021-05-01
2021-05-28
```

* Java I/O API:

→ Java I/O is a set of classes that give access to external resources including file systems and the network.

→ This Java i/o API, presenting the basic notions you need to understand in order to start writing java code that takes advantage of the API.

1. understanding the Main Java I/O concepts:

→ Introducing the Java I/O API: files, Streams, paths, Streams of character and Streams of bytes.

2. File System Basics

→ working with the file System: working with files and paths, working with directories, getting file metadata

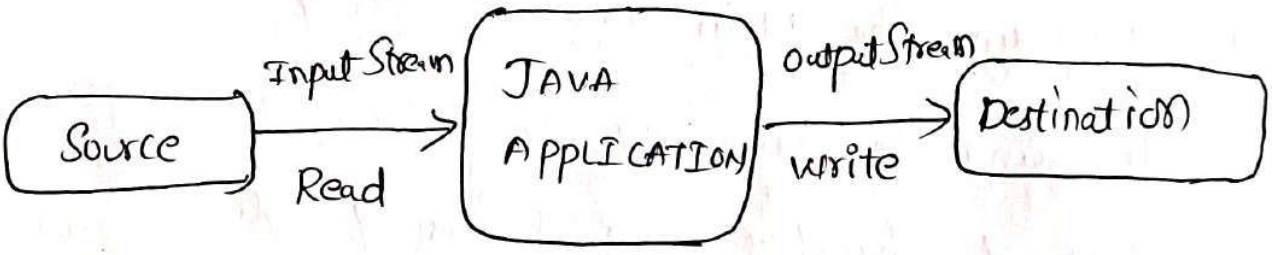
3. File Operation Basics

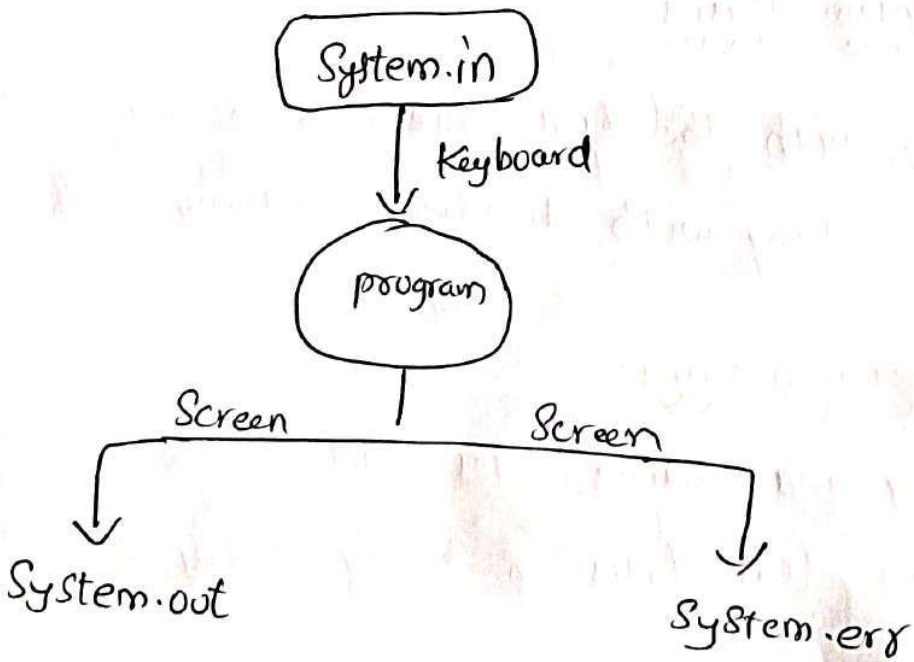
→ writing and reading files: getting to know the difference between text files & binary files.

Input-output in Java with Examples:

→ Java brings various Streams with i/o package that helps the user to perform all the input-output operations.

→ These Streams support all the types of objects, datatypes, characters, files etc to fully execute the I/O operations.





Standard I/O Streams in java

System.in:

→ This is the Standard input Stream (System.in) that is used to read charaitery from the keyboard (or) any other Standard input device.

System.out:

→ This is the Standard output Stream (System.out) that is used to produce the result of a program on an output device like the computer screen.

print():

→ This method in java is used to display a text on the Console. This text as the parameter to this method in the form of a String.

Syntax:

```
System.out.print(parameter);
```

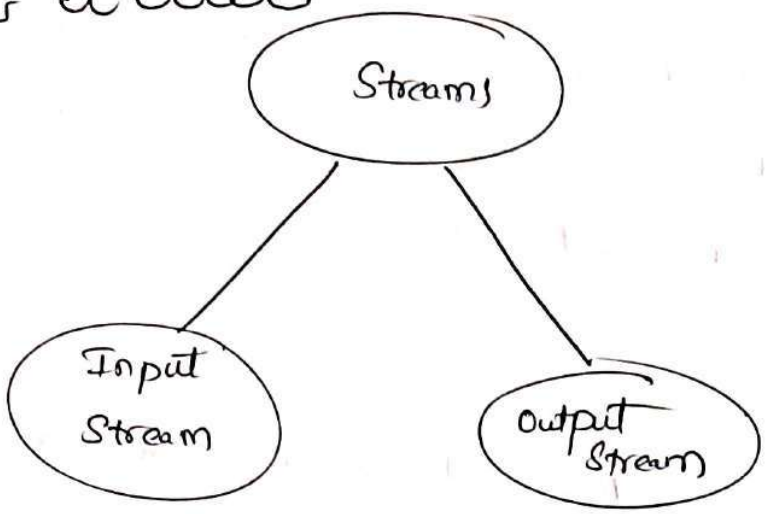
Ex:

```
import java.io.*;
Class Demo_Print
{
    psvm(String args[])
    {
        S.o.p("Kalam");
        S.o.p("Kalam");
        S.o.p("Kalam");
    }
}
```

O/p:

```
Kalam Kalam Kalam
```

* Types of Streams:

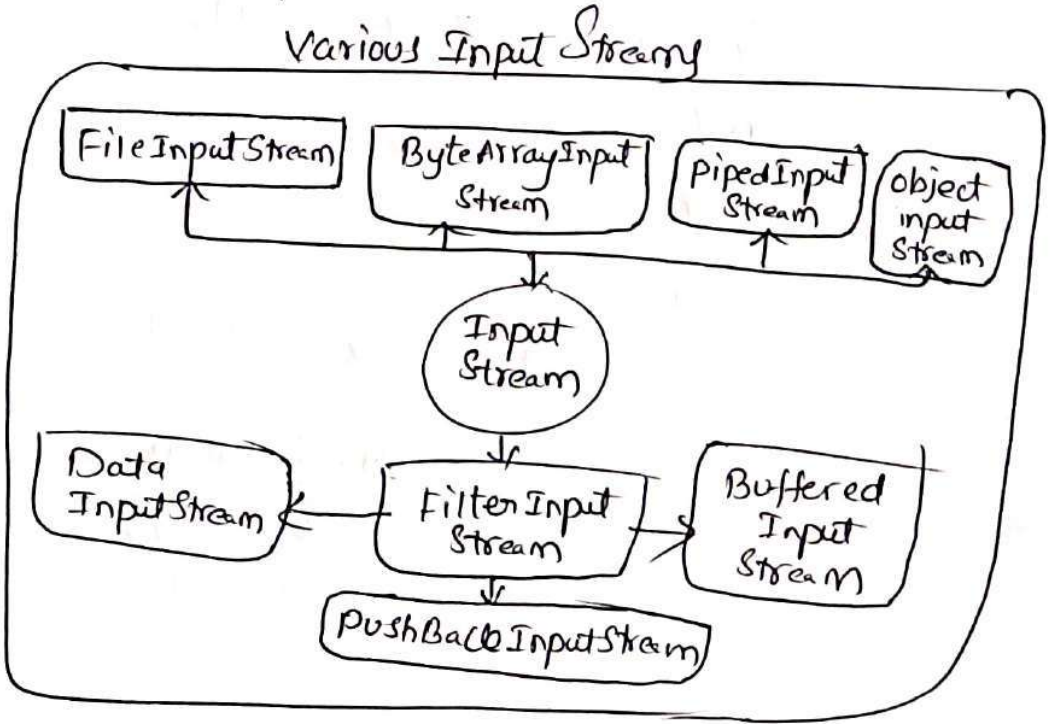


Depending on the type of operations, Streams can be divided into two primary classes.

Input Stream:

→ These Streams are used to read data that must be taken as an input from a source array (or) file (or) any peripheral device.

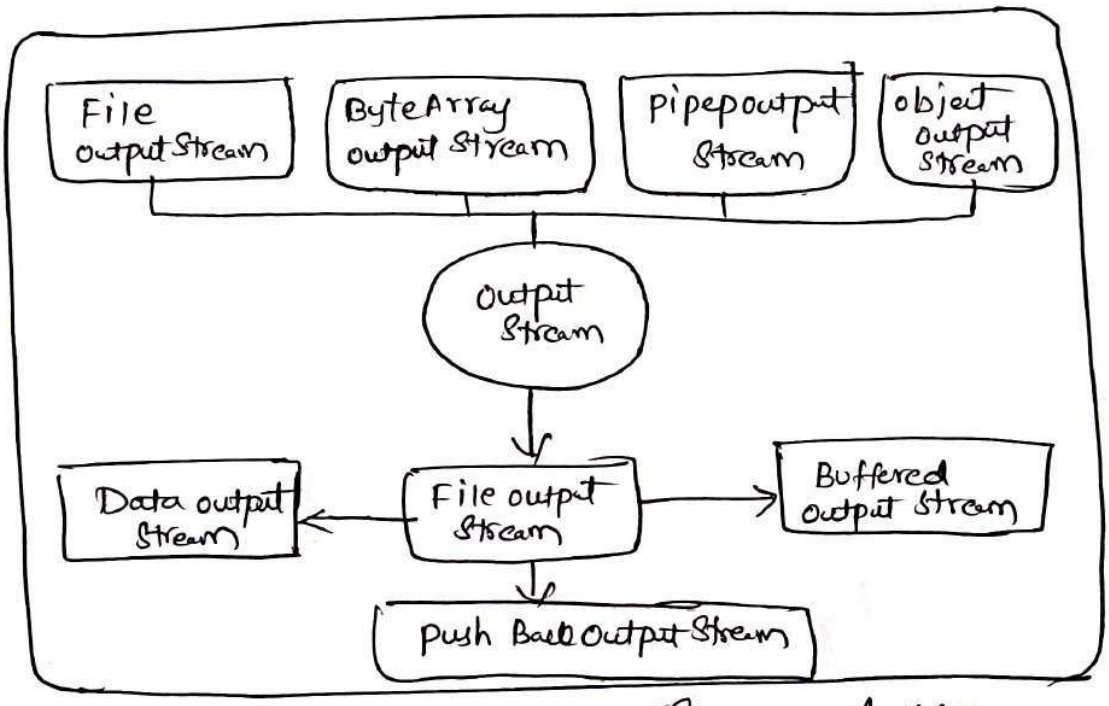
Ex: FileInputStream, BufferedInputStream etc



OutputStream:

→ These Streams are used to write data as outputs into an array (or) file (or) any output peripheral device.

Ex: FileOutputStream, BufferedOutputStream etc.



Various Output Stream classes

Example

write Java Input & output Stream programs

// Storing string program [File output Stream program]

```
import java.io.*;

class FileDemo
{
    psvm (String args[])
    {
    try
    {
FileOutputStream fout = new FileOutputStream ("D:
                                        AB.txt");

String S = "Paramesh Java TUTorials";

byte b[] = S.getBytes();

fout.write(b);
fout.close();
S.o.pln("Success");
}
catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

O/P

=

Paramesh Java TUTorials

// Single Character (Read)

File AB.txt

Kakinada

```
import java.io.*;

Class FileDemo
{ psvm (String args[])
  {
  try
  {
  FileInputStream fis = new FileInputStream("D:\\AB.txt");
  int i = fis.read();
  S.o.pln((char)i);
  fis.close();
  }
catch (Exception e)
{ e.printStackTrace();
}
}
}
```

o/p:

K

// String (Read) [File InputStream program]

```
import java.io.*;

Class FileDemo
 {
   psvm (String args[])
     {
   try {
     {
FileInputStream fis = new FileInputStream ("D:AB.txt");
   int i=0;
   while ((i!=fis.read())!= -1);
    }
     fis.close();
   }
   catch (Exception e)
     {e.printStackTrace();
       }
     }
   }
```

AB.txt

JavaTutorial

O/p

Java Tutorial

Input to programs: [Scanner class]

Java Input(System.in)

→ Java provides different ways to get input from user where Scanner class is one of the important one.

→ In order to use Scanner class we need to write a import Statement at top of the program.

In order to use Scanner class lets consider following:

(i) import a package

→ import java.util.*;

(ii) Construct Scanner class object

→ Scanner sc = new Scanner(System.in);

(iii) Define a variable to receive a value.

int x;

(iv) Read input using below methods

Method	Description
nextByte();	Accepts a Byte
nextShort();	Accepts a Short
nextInt();	Accepts an int
nextLong();	Accepts a long
next();	Accepts a single word
nextLine();	Accepts a line of String
nextfloat();	Accepts a ~~double~~ float
nextBoolean();	Accepts a boolean

Program:

```
/* write a java program Addition of two number
using Scanner class */
import java.util.*;
Class Addition
{
public Static void main (String args[])
{
Scanner sc = new Scanner(System.in);
int x,y,z;
System.out.print("Enter 1st no:");
x = sc.nextInt();
System.out.print("Enter 2nd no:");
y = sc.nextInt();
z = x+y;
System.out.println("TOTAL = "+z); } }
```

Output:

```
Enter 1st no : 5
Enter 2nd no : 10
TOTAL = 30
```

* Class String:

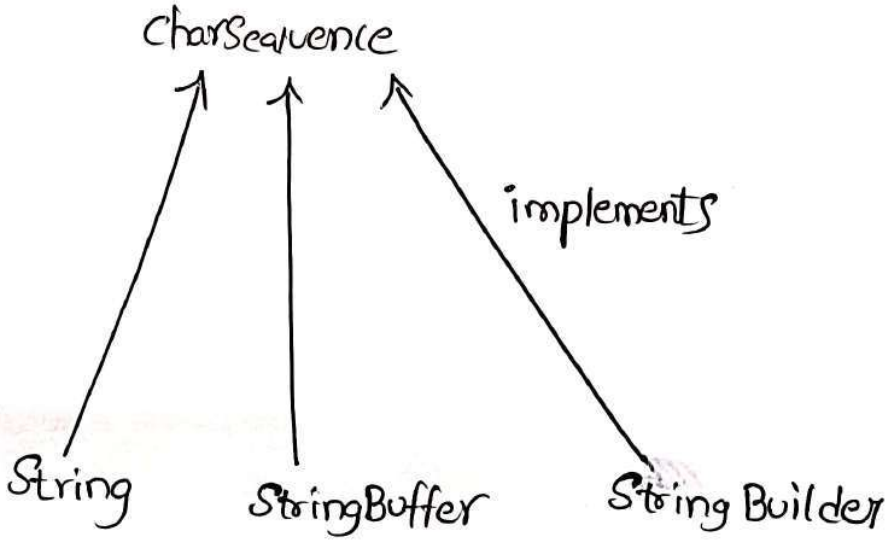
→ The java.lang.String Class provides many useful methods to perform operations on sequence of char values.

S.NO	Method	Description
1.	char charAt(int index)	→returns char value for the particular index.
2.	int length()	→returns String length
3.	String subString(int beginIndex)	→returns substring for given begin index.
4.	boolean contains(CharSequence s)	→ returns true (or) false after matching
5.	boolean isEmpty()	→check if String is empty.
6.	String Concat(String str)	→ concatenates the specified String.
7.	String toLowerCase()	→ returns a String in lowercase.
8.	String toUpperCase()	→ returns a string in uppercase.
9.	String trim()	→ removes beginning and ending spaces of this String
10.	int indexOf(String substring)	→ returns the specified Substring index.

* Interface charSequence:

→ The charSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it.

→ It means, we can create strings in Java by using these three classes.



→ The Java String is immutable which means it cannot be changed. whenever we change any String, a new instance is created.

→ For mutable Strings, you can use StringBuffer and StringBuilder classes.

* String Buffer:

→ StringBuffer class is used to create mutable String objects.

→ StringBuffer class is the same as String class except it is mutable .i.e. It cannot be changed.

→ String and String Buffer classes are predefined classes in Java.lang.package.

Creation of StringBuffer

Syntax

```
StringBuffer  reference variable = new StringBuffer ("String");
                                   (operator)   (constructor)
```

Ex

```
StringBuffer  sb  = new StringBuffer ("Kalam Sir");
```

* Program

```
import java.lang.*;
Class SB
{
    psvm (String args[])
    {
String Buffer st = new StringBuffer("KalamSir");
S.o.pln(st);
    }
}
```

Output

KalamSir

* String Buffer Methods (or) Methods for extracting characters from Strings

(1) append():

→ It is used to append the specified string with the existing string.

Ex:

```
StringBuffer sb = new StringBuffer("welcome");
sb.append("Java");  → o/p welcome Java
```

(2) insert():

→ It is used to insert the specified string with the string at the specified position.

Ex:

```
sb.insert(4, "rama");  o/p → welcramaomeJava
```

(3) replace():

→ It is used to replace the string from specified startIndex and endIndex.

Ex: `sb.replace(4,7,"sita");` → o/p welcsitaomeJava

(4) delete():

→ It is used to delete the String from specified Startindex and endindex.

Ex:

sb.delete(4,7); → O/P welcomeJava

(5) length():

→ It is used to return the character at the specified position length of the String.

Ex:

sb.length(); → O/P 11

(6) reverse()

→ It is used to reverse the String.

Ex:

sb.reverse(); → O/P avaJemoclew

(7) charAt():

→ It is used to return the character at the specified position.

Ex

sb.charAt(5); → O/P : m

(4) delete():

→ It is used to delete the String from specified startindex and endindex.

Ex:

sb.delete(4,7); → O/P welcomeJava

(5) length():

→ It is used to return the character at the specified position length of the String.

Ex:

sb.length(); → O/P 11

(6) reverse()

→ It is used to reverse the String.

Ex:

sb.reverse(); → O/P avaJemoclew

(7) charAt():

→ It is used to return the character at the specified position.

Ex

sb.charAt(5); → O/P : m

(8) subString():

→ It is used to return the substring from the specified beginIndex.

Ex:

~~Sbeindex~~

sb.subString(7); → O/P Java

(9) indexOf():

→ It return the specified substring index.

Ex

sb.indexOf('e'); → O/P: 1

* Program:

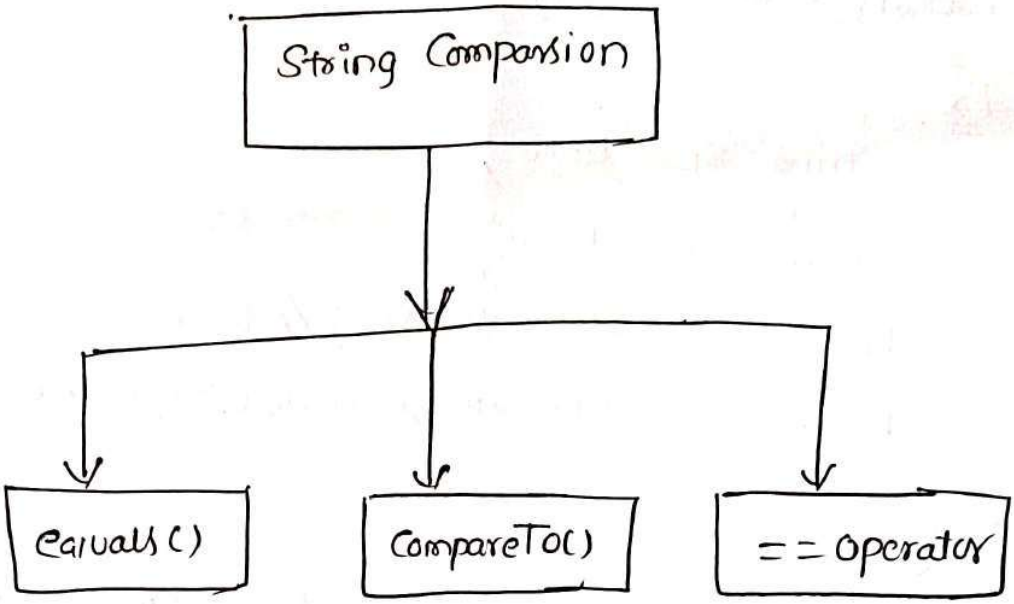
```
import java.lang.*;

    class StringBufferDemo
     {
        psvm(String args[])
     {
StringBuffer sb = new StringBuffer ("welcome");

        sb.append("Java"); o/p → welcomeJava
        sb.insert(4, "rama"); o/p → welcramaomeJava
        sb.replace(4, 7, "sita"); o/p → welcsitaJava
        ~~sb.delete(4,7);~~ → ~~welcomeJava~~
        sb.length(); o/p → 11
        sb.reverse(); o/p → avaJ emoclew
        sb.charAt(5); o/p → m
        sb.indexOf('e'); o/p → 1
        sb.subString(7); o/p → Java
     }

}
```

* String Comparsion in Java:



In Java we can compare String Comparsion into three ways.

(1) equals()

(2) CompareTo()

(3) == operator

(1) equals():

→ equals() method is mainly useful inorder to compare whether the contains of two Strings are equal (or) not.

→ If the contains of two Strings are Equal then it returns true value.

→ If the contains of two Strings are not Equal then it returns false value.

Syntax

```
boolean equals(String);
```

Ex:

```
String s1 = "Hello";
String s2 = "hello";
boolean a = s1.equals(s2); // false
boolean a = s1.equalsIgnoreCase(s2); //true
```

(2) compareTo():

→ compareTo() method compares two Strings. If both the Strings are Equal then it returns the value "zero".

→ If string S_1 greaterthan String S_2 i.e. $(S_1 > S_2)$ it gives +ve value.

→ If string S_1 less than String S_2 $(S_1 < S_2)$ it give -ve value.

Syntax

```
int compareTo(String);
```

Ex

(1)
```
String s1 = "hai";
String s2 = "hai";
int a = s1.compareTo(s2); // a=0
```

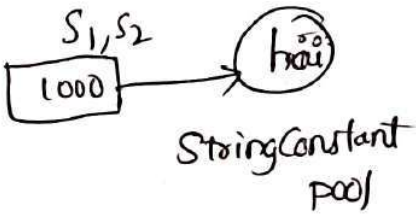
Ex 2

```
String s1 = "hai";
String s2 = "Hai";
int a = s1.compareTO(s2); // a=32
int a = s1.compareTOignoreCase(s2); // a=32
```

a = 97
A = 65
97
-65
32

(3) == operator:

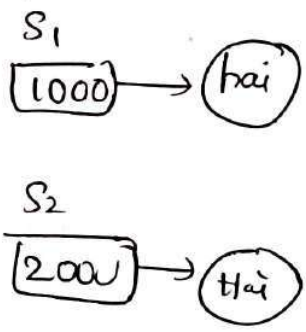
→ It is used for reference matching i.e. whether both objects are same (or) not.



Ex:

(1)
```
String s1 = "hai";
String s2 = "hai";
s.o.p(n (s1==s2); // True
```

(2)
```
String s1 = new String("hai");
String s2 = new String("hai");
s.o.pln(s1==s2); // false
```



* Inter Thread Communication in Java (or) Thread class methods in Java (or) wait() and notify()

→ Inter thread communication allows us one Synchronized thread can communicate with other Synchronized thread.

→ Inter thread communication is a mechanism in which a thread goes into wait state until another thread sends a notification.

wait():

→ A thread goes into Sleeping state until some other thread calls notify().

notify():

→ It wakes up a thread that called wait() on same object.

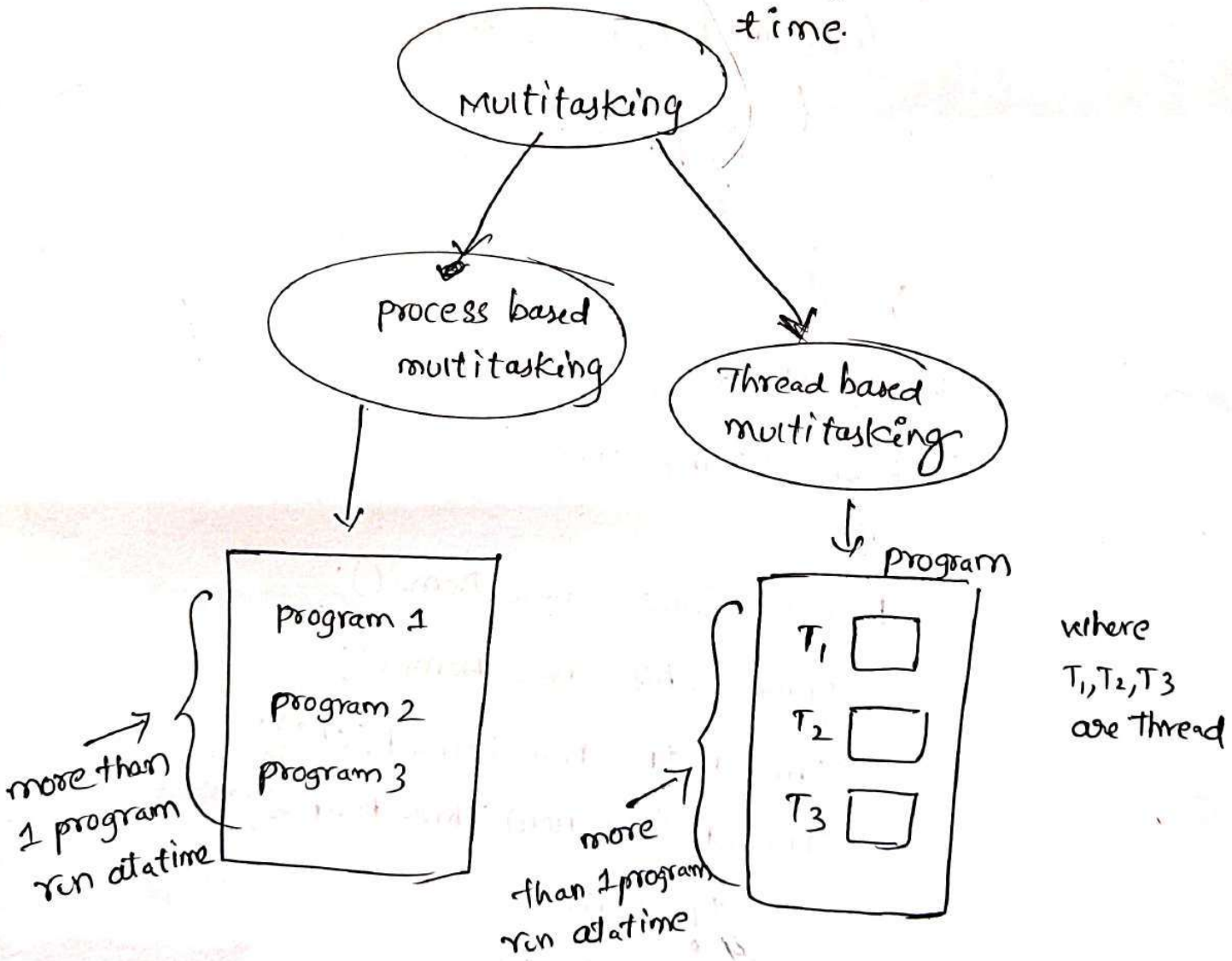
Ex

```
class Sample
{ psvm(String args[])
  {
    Mythread t = new Mythread();
    t.start();
    Synchronized(t)
    { t.wait();
      s.o.pln(t.total); }
```

* Multithreading :- (Introduction)

Multitasking – means performing multiple tasks simultaneously.

Simultaneously
└ means happening (or) being done at same time.



Thread :

→ More than one threads runs at a time

→ Executing more than one threads of a same program is called "Multithread"

* Program:

```
import java.lang.*;
Class Demo extends Thread
{
  public void run()
  {
    for(int i=1; i<=5; i++)
    {
      S.o.pln(i);
    }
  }
}
Class MDemo
{
  psvm(String args[])
  {
    Demo ob1 = new Demo();
    Demo ob2 = new Demo();
    Thread t1 = new Thread(ob1);
    Thread t2 = new Thread(ob2);
    t1.start();
    t2.start();
  }
}
```

Olp:

1
1
2
2
3
3
4
4
5
5

* String Builder class:

→ String Builder class has been added in JDK 1.5 which has same features like String Buffer class.

→ String Builder class objects are also ~~mutable~~ mutable as the String Buffer objects.

Difference.

→ String Buffer is class is Synchronized and String Builder is not.

* Multithreaded programming:

Introduction, Need for multiple Threads Multithreaded Programming for multi-core processor:-

→ Multithreading in Java is a process of executing multiple thread Simultaneously.

→ A thread is a light weight Sub-process, the Smallest unit of processing. Multiprocessing & multi-threading, both are used to achieve multi tasking.

Advantages of Java Multithreading:

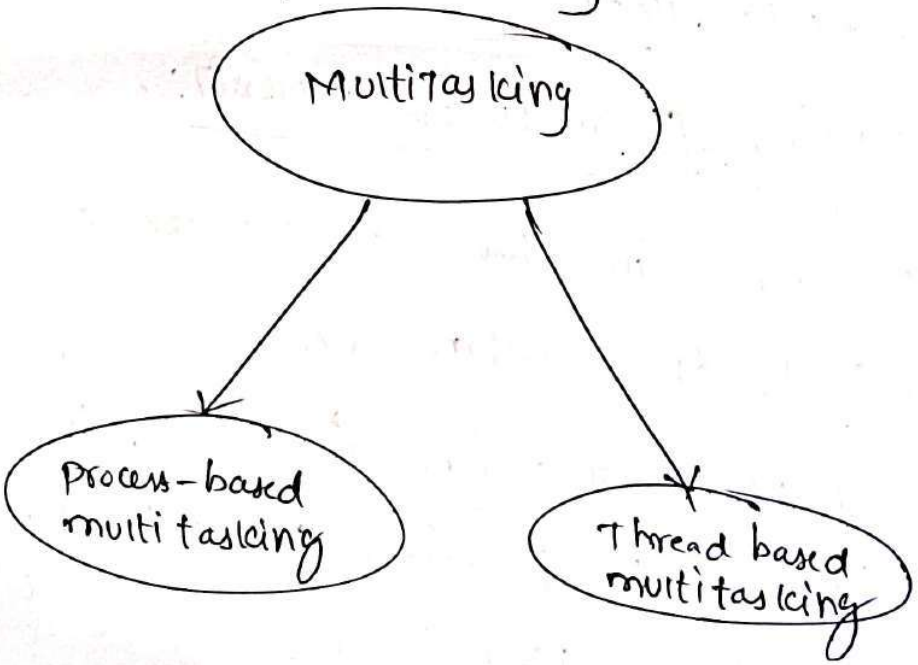
→ It doesn't block the user because threads are independent & you can perform multiple operations at the same time.

→ you can perform many operations together, so it save time.

→ Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

* Multi Tasking:

→ Multitasking is a process of executing multiple tasks simultaneously. we use multitasking to utilize the CPU. Multitasking can achieved in two ways



(i) Process-based Multitasking (Multi-processing)

→ Each process has an address in memory. In other words, each process allocates a separate memory area.

→ A process is heavy weight.

→ cost of communication between the process is high.

→ Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists etc.

(ii) Thread based Multitasking (Multithreading)

→ A Thread share the same address space.

→ A Thread is light weight

→ cost of communication between the thread is low.

* Thread class :

→ Thread class provide constructors and methods to create and perform operations on a thread.

→ Thread class extends object class & implements Runnable interface.

Commonly used constructors of Thread class

(1) Thread()

(2) Thread (String name)

(3) Thread (Runnable r)

(4) Thread (Runnable r, String name)

Commonly used methods of Thread class :

(1) public void run() :- is used to perform action for a thread.

(2) public void start() :- starts the execution of the thread. JVM calls the run() method on the thread

(3) public void join() : waits for a thread to die.

(4) public void sleep(long milliSeconds): Cause the currently executing thread to sleep for the specified number of milliSeconds.

(5) public int getPriority(): returns the priority of the thread.

(6) public int setPriority(int priority): changes the priority of the thread

(7) public int getId():

→ returns the id of the thread.

(8) public void suspend():

→ is used to suspend the thread.

(9) public void resume():

→ is used to resume the suspended the thread

(10) public void stop():

→ is used to stop the thread.

(11) public void interrupt():

→ interrupts the thread.

* Thread Creation (or) Creation Of New Threads

→ To create a thread we can follow below step.

Step1 :

→ To create a class that extends thread class (or) Runnable interface. Ex : Class Demo extends Thread

Step2 : (or) class Demo implements Runnable

→ write run() method with body.

Step3 :

→ create an object to the class and attach to a thread.

Ex :

```
Demo ob1 = new Demo();
Thread t1 = new Thread (ob1);
```

Step4 :

→ Start the thread using Start() method.

Ex : t1.Start();

// Example program of Thread Creation

```
import java.lang.*;

Class Demo extends Thread
{
    public void run()
    {
        S.o.pln("Hello");
    }
}

Class MDemo
{
    psvm(String args[])
    {
        Demo ob = new Demo();

Thread t = new Thread(ob);
        t.start();
    }
}
```

O/P

Hello

* Main Thread - Creation of New Threads:

→ A Thread represents a Separate path of execution.

→ Group of Statements executed by JVM one by one.

program to find the thread used by JVM to execute the Statements (Main Thread)

```
import java.lang.*;

class ThreadName
{
  psvm(String args[])
  {
    S.o.pln("Welcome kalam");

    Thread t = Thread.currentThread();

    S.o.pln("current Thread is :"+t);

    S.o.pln("current Thread Name is :"+t.getName());
  }
}
```

→ In the above program currentThread()
is a Static method of Thread class, which returns the reference of the current running thread.

→ Here Thread indicates that 't' is thread class object.

* Life cycle of Thread (or) States of a thread

Thread life cycle have five states

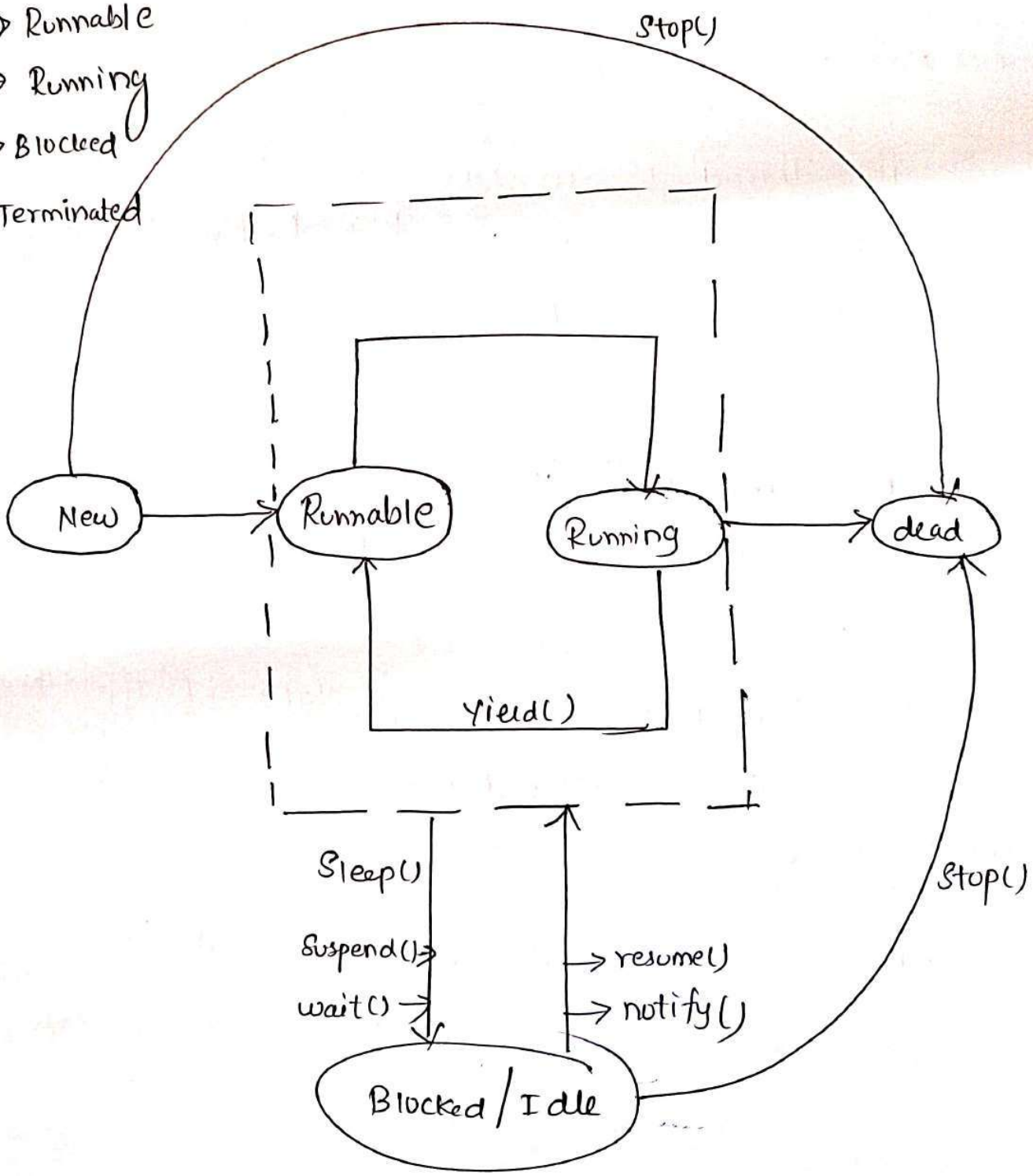
→ New

→ Runnable

→ Running

→ Blocked

→ Terminated



→ The life cycle of the thread in java is controlled by JVM.
The java thread states are as follows

(1) New:

→ The Thread is in new state if you create an instance of thread class but before the invocation of start() method.

(2) Runnable:

→ The Thread is in runnable state after invocation of start() method.

(3) Running:

→ The Thread is in running state if the thread scheduler has selected it.

(4) Blocked:

→ This is the state when the thread is still alive, but is currently not eligible to run.

(5) Terminated:

→ A thread is in terminated (or) dead state when its run() method exists.

* Thread Control methods in Java:

→ Core java offers complete control over multithreaded program such as suspending thread, resuming thread, or stopping thread completely based on requirements.

→ These methods are used to control the thread's behavior.

public void suspend():

→ Moves a thread into the suspended state, and can be resumed using resume() method.

public void stop():

→ Stops a thread completely.

public void resume():

→ Resumes a thread, which is suspended using Suspend() method.

public void wait():

→ Makes the current thread to wait until another thread invokes the notify().

public void notify()

→ Wakes up a single thread that is waiting on this object's monitor.

* Program:

```
import java.lang.*;
Class ThreadDemo extends Thread
{
  public void run()
  {
  try
    {
      for(int i=0;i<3;i++)
       {
S.o.p(Thread.currentThread().getName()+i);
  Thread.Sleep(1000);
    }
  } Catch(InterruptedException e)
    {
      e.printStackTrace();
    }
    }
Public class Main
  {
    psvm(String args[])
      {
ThreadDemo t1 =newThreadDemo();
  t1.SetName("First Thread:");
ThreadDemo t2 = new ThreadDemo();
  t2.SetName("Second Thread");
```

```
ThreadDemo t3 = new ThreadDemo();
    t3.setName("Third thread");
        t1.start(); // call run() method
        t2.start();
        t2.suspend(); // Suspend t2 thread
        t3.start(); // call run() method
        t2.resume(); // resume t2 thread
        }
    }
```

Output

```
FirstThread : 0
Third Thread : 0
Second Thread : 0
First Thread : 1
Third Thread : 1
Second Thread : 1
First Thread : 2
Third Thread : 2
Second Thread : 2
```

* Thread Synchronization :

→ when multiple threads are acting on same java object then there may be a chance of getting data inconsistency problem.

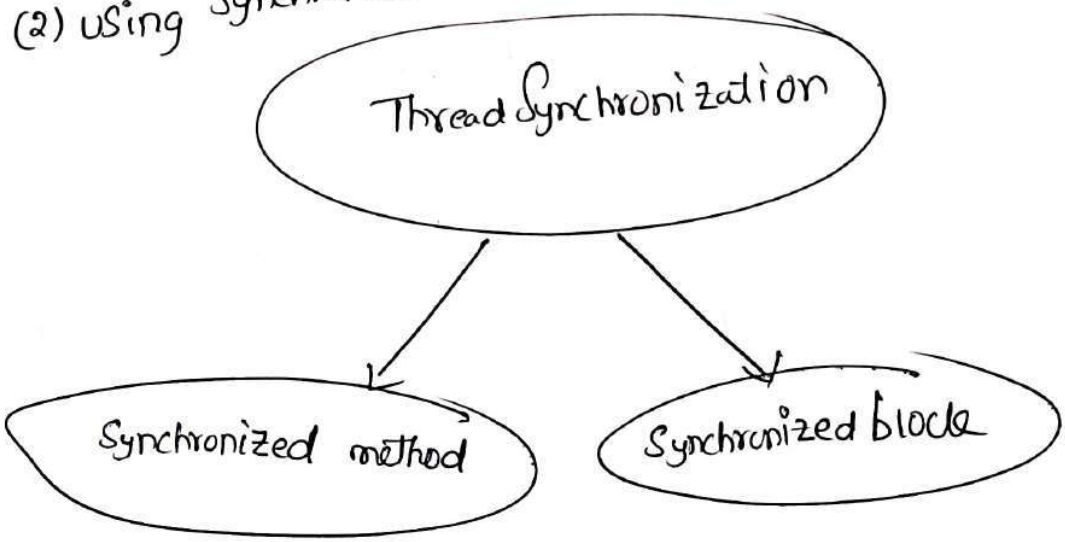
→ To overcome that problem we go for Thread Synchronization concept.

→ when a thread is already acting on an object, preventing any other thread from acting on the same object is called Thread Synchronization (or) thread safe.

There are two ways to achieve Synchronization in java.

(1) using Synchronized method

(2) using Synchronized block



(1) Using Synchronized method:

→ we can Synchronize an entire method by using Synchronized keyword.

Syntax:

```
Synchronized method-name (par-list)
{
    Body of the method
}
```

(2) Using Synchronized block:

→ If very few lines of the code required Synchronization then we should go for Synchronized block.

Syntax:

```
Synchronized (this)
{
----
}
```

* Thread priorities:

→ Every thread in java has some priority. it may be default priority given by JVM (or) customized priority provided by programmer.

→ The valid range of thread priorities is from 1 to 10.

→ where 1 is min priority and 10 is max priority.

→ Thread class defines the following constants to represent some standard priorities.

- ↳ Thread.MIN_PRIORITY
- ↳ Thread.NORM_PRIORITY
- ↳ Thread.MAX_PRIORITY

→ thread class defines the following methods to get and set priority of a thread.

- ↳ public final int getPriority()
- ↳ public final void setPriority(int p);

* Multithreading

→ It is a programming concept in which a program divided into two (or) more sub programs (or) threads that are executed at same time in parallel.

→ Executing multiple threads simultaneously is known as multithreading.

Uses of multithreading:

→ used to develop multimedia graphics.
→ used to develop animation.
→ used to develop video games.
→ used to develop web servers and application servers

* Deadlock is Race Situation:

→ Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread.

→ Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

→ Multithreaded programming in java suffers from the deadlock situation because of the Synchronized keyword.

* Program:

```
public class DeadlockDemo
{
  psvm(String args[])
    {
    final String r1 = "R1";
    final String r2 = "R2";
  Thread t1 = new Thread()
    {
      public void run()
        {
          Synchronized (r1)
          {
           S.o.pln("Thread 1 acquired lock
                                on Resource 1");
          try
          {
            Thread.Sleep(1000);
          }
```

```
catch(Exception e)
    { s.o.pln(e);
    }
    Synchronized (r2)
    { s.o.pln ("Thread 1 acquired
      }
    }
  }
Thread t2 = new Thread()
  { public void run()
    {
      Synchronized (r2)
       { s.o.pln ("
      try
      {
        Thread.Sleep(1000);
      }
      catch (Exception e)
        { s.o.pln (e);
        }
        Synchronized (r2)
        { s.o.pln ("Thread 1
        } } }
```

* Inter Thread Communication in Java (or) Thread class methods in java (or) wait() & notify()

(or)

Suspending, Resuming and Stopping of Threads

→ Inter Thread Communication allows us one Synchronized thread can communicate with other Synchronized thread.

→ Inter thread Communication is a mechanism in which a thread goes into wait state until another thread ~~Synchronized~~ Sends a notification.

wait():

→ A thread goes into Sleeping state until some other thread calls notify()

notify():

→ It wakes up a thread that called wait() on Same object.

Example

```
import java.lang.*;

Class Sample
{
  psvm (String args[])
  {
    MyThread t = new MyThread();
      t.start();
      Synchronized (t)
      {
        t.wait();
        S.o.pln(t.total);
      }
    }
  }

Class MyThread extends Thread
  {
    int total=0;
    public void run()
    { Synchronized (this)
      {
        for(int i=1; i<=10; i++)
          total=total+i;
        this.notify();
      } } }
```

* Thread Class Methods in Java

Stop() :-

→ It Stops execution of currently running thread.

Ex:

```
import java.lang.*;
class mythread extends Thread
{
public void run()
{
for(int i=1; i<=5; i++)
{ if(i==3)
{
Stop(); //Static method
}
S.o.pln("In mythread i=" +i);
}
S.o.pln("Exit from mythread");
}
}
class DemoStop
{ psvm(String args[])
{
Mythread t = new mythread();
t.start();
for(int j=1; j<=5; j++)
S.o.pln("In Main thread=" +j);
S.o.pln("Exit from main method"); } }
```

O/P

```
In Main thread j=1
In mythread i=1
In mythread i=2
In mainthread j=2
In main thread j=3
In main thread j=4
In main thread j=5
```

executed from main method

```
class mythread extends thread
{
    int total = 0;
    public void run()
    {
    Synchronized (this)
    {
      for(int i=1 ; i<=10; i++)
          total = total + i;
          this.notify();
      }
    }
```

Sum of 'n' natural numbers

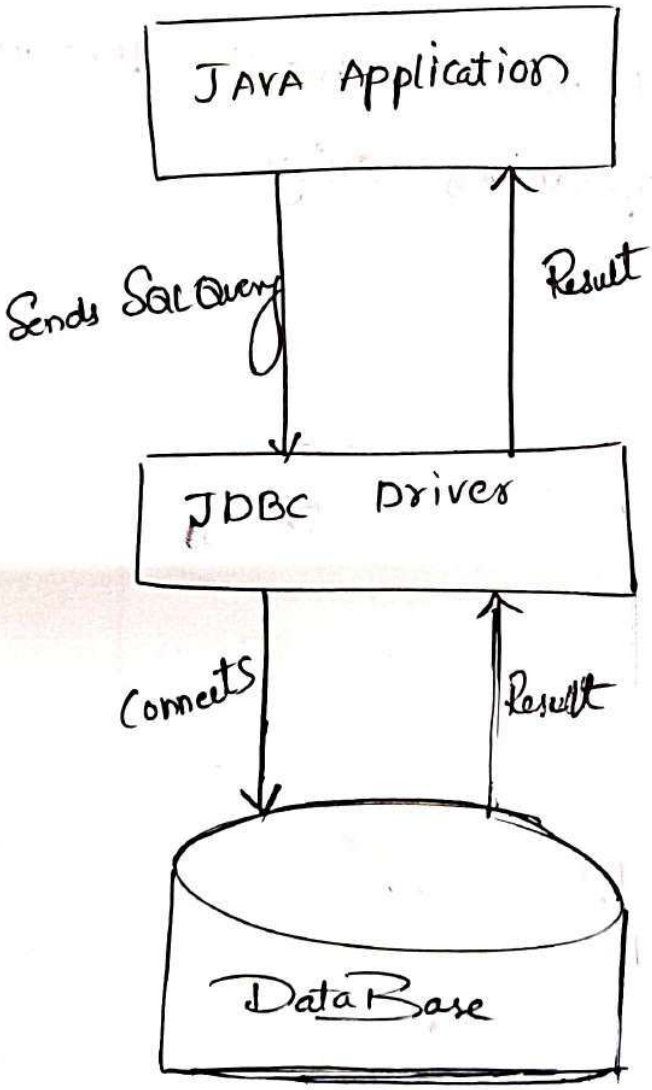
$\frac{10 \times 10+1}{2} = 55$

* ~~thread class~~

* Introduction to JDBC:

→ JDBC Stands for "Java Data Base Connectivity".

→ It is an API(Application programming Interface) that defines how a java programmer can access the database in tabular format from java code.



Working of JDBC

Features of JDBC:

→ JDBC is an API, using which we can communicate with any database without rewriting our java application.

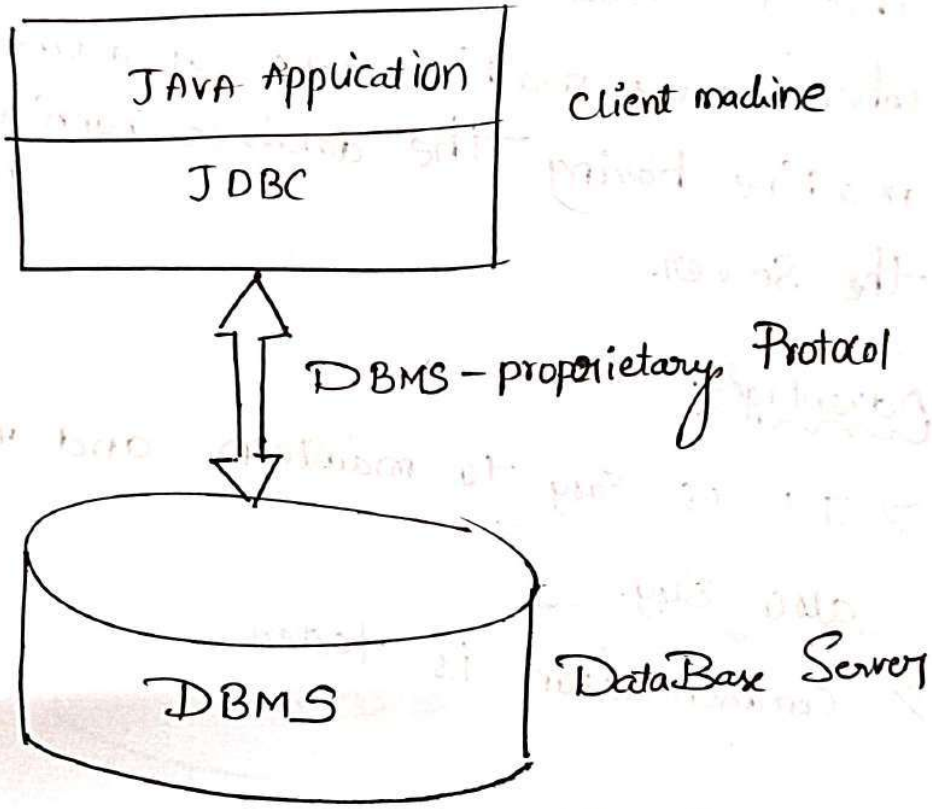
→ Most of the JDBC drivers are implemented in java. Hence JDBC is known as platform independent technology.

→ Using JDBC API, we can able to perform all basic database operations easily.

* JDBC Architecture:

→ The JDBC API supports both two-tier and three-tier processing models for database access.

Two-tier Architecture:



→ In the two-tier model, a java application communicates directly with the database.

→ For this purpose it requires a JDBC driver to communicate with the particular database being accessed.

→ After that the user sends a Query to the database.

→ The database process the Query and then sends the response back to the user.

→ In this model, the database may not always present in a single machine.

→ It can be located on a different machine on a network to which a user is connected.

→ This is known as client-server configuration. where user machine acts as a client and the machine having the database running acts as the server.

Advantages:

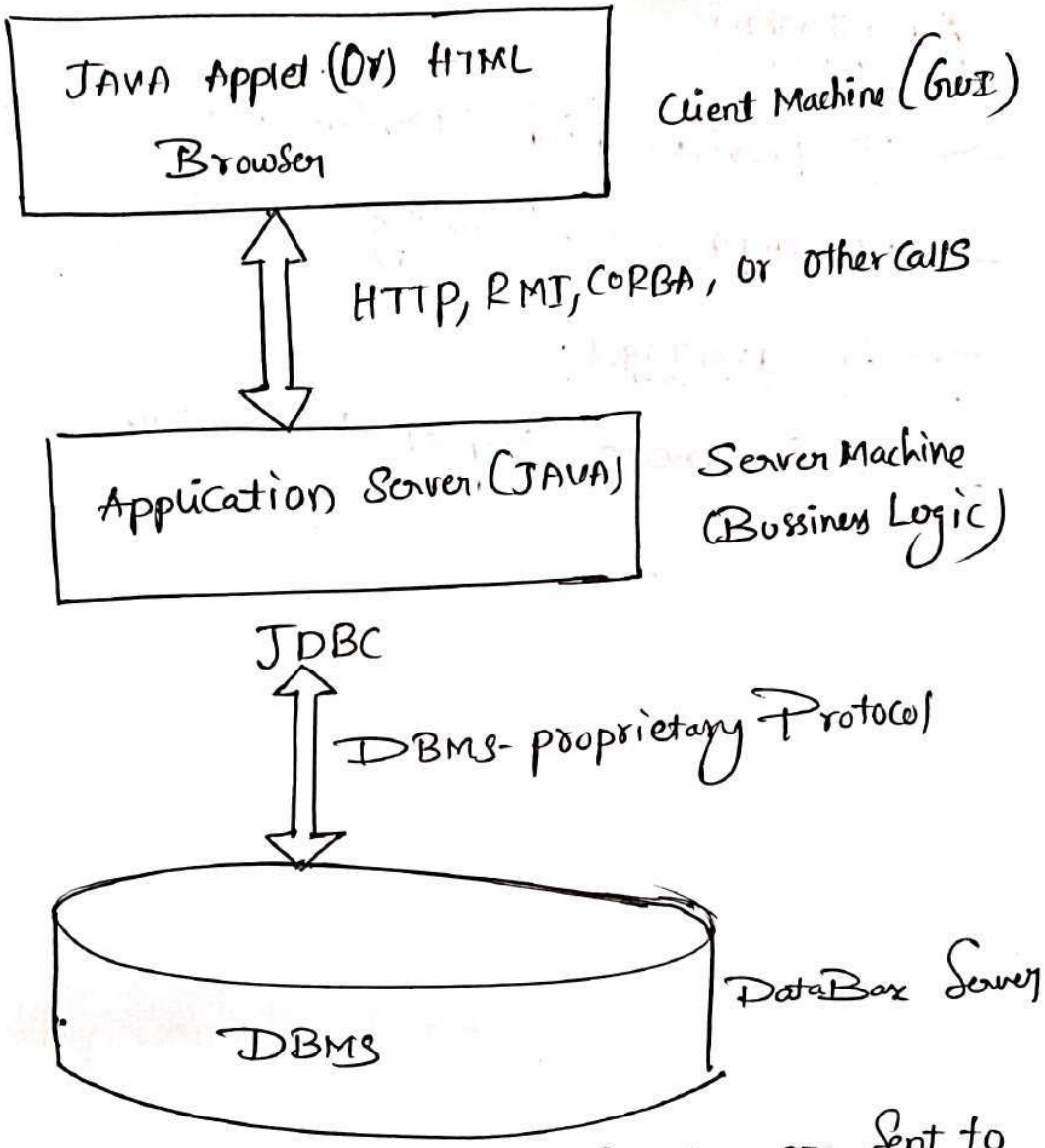
→ It is easy to maintain and modification also easy.

→ Communication is faster.

Dis-advantages:

→ In this architecture, application Performance will be degrade based on increasing the users.

* Three-Tier Architecture:



→ In three-tier model, the user Queries are sent to the middle tier Server.

→ From middle tier Server the queries are again sent to the database.

→ The database takes the Queries from middle tier Server & process those Queries and then sends the response back to the middle tier Server.

→ After that, middle tier Server Sends the results back to the User. *

Advantages:

→ It provides high Performance.

→ It improves Security.

Dis-advantages:

→ It is more complex to maintain

* Establishing JDBC Database Connections:

Steps for writing a JDBC Program

Step 1: Register the driver (or) loading the driver

→ To load (or) register the driver we can use any one of the following methods.

Syntax:

```
Class.forName("url");
DriverManager.registerDriver("url");
System.setProperty("url");
```

Step 2: Establishing the connection to database

→ In this step we establish a connection with a specific database. The connection establishment can be done by Connection Interface.

Syntax:

```
Connection con = DriverManager.getConnection("url");
```

Step 3:

→ In this step we should create SQL statement in our java program using any one of the interface like Statement (or) prepared Statement (or) CallableStatement.

Syntax:

```
Statement st = con.createStatement();
preparedStatement PS = con.prepareStatement("Sql Query");
```

Step4: Executing the SQL statement

→ For this purpose we use execute(), executeQuery() and executeUpdate() methods of Statement interface.

Syntax:

```
Boolean b = St.execute("Sql Query");
int i = St.executeUpdate("insert or delete or update Query");
ResultSet rs = St.executeQuery("select Query");
```

Step5: Retriving the Results

→ The Results obtained by executing the Sql Statement can be stored in an object with the help of ResultSet interface.

Syntax:

```
ResultSet rs = St.executeQuery("Select * from table name");
```

→ ResultSet Object maintain a cursor.

→ When ResultSet object is first created, then the cursor is pointed before the first row of table.

→ In order to move the cursor to next row we need to use next method.

Step 6: Close the Connection

→ To close the connection between Java program and database we use one method close() of connection interface.

Syntax:

```
con.close();
```

* Result Set Interface:

→ object of Result set interface maintains a cursor pointing to a row of table.

Commonly used Methods:

→ boolean next() → True (row present)
                 → False (no data)

→ boolean previous()

→ boolean first → first row in resultset object

→ boolean last () → last row.

:

:

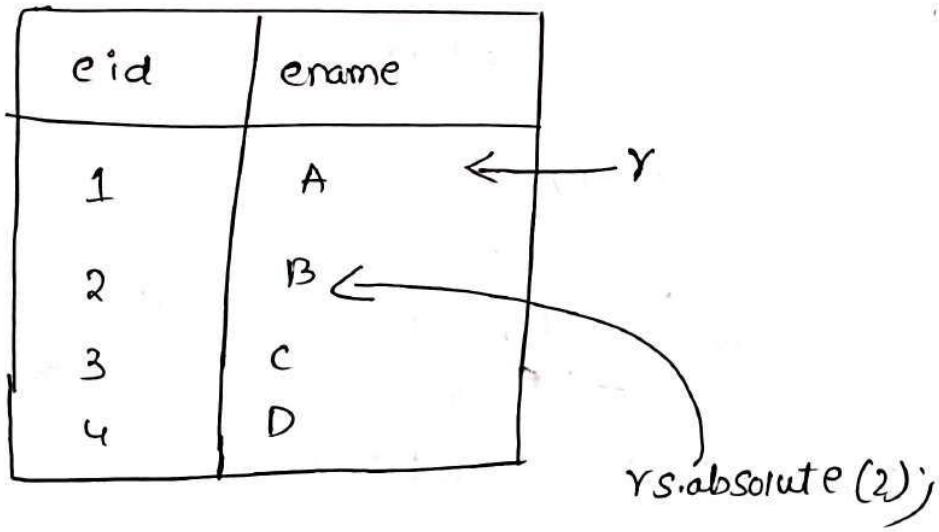
boolean absolute (int row) → uth row

rs.absolute(4);

Example:

```
Start
ResultSet rs = Start.executeQuery("select * from emp");
    while (rs.next())
    {
System.out.println(rs.getString(1) + " " + rs.getString(2));
    }
```

emp [ResultSet object]



* Java FX GUI :

Introduction:

→ JavaFX is a Java library used to build Rich Internet Applications.

→ The applications written using this library can run consistently across multiple platforms.

→ The applications developed using JavaFX can run on various devices such as
  - Desktop computers
  - mobile phones
  - TVs
  - Tablets etc.,

→ To develop GUI applications using Java programming language.

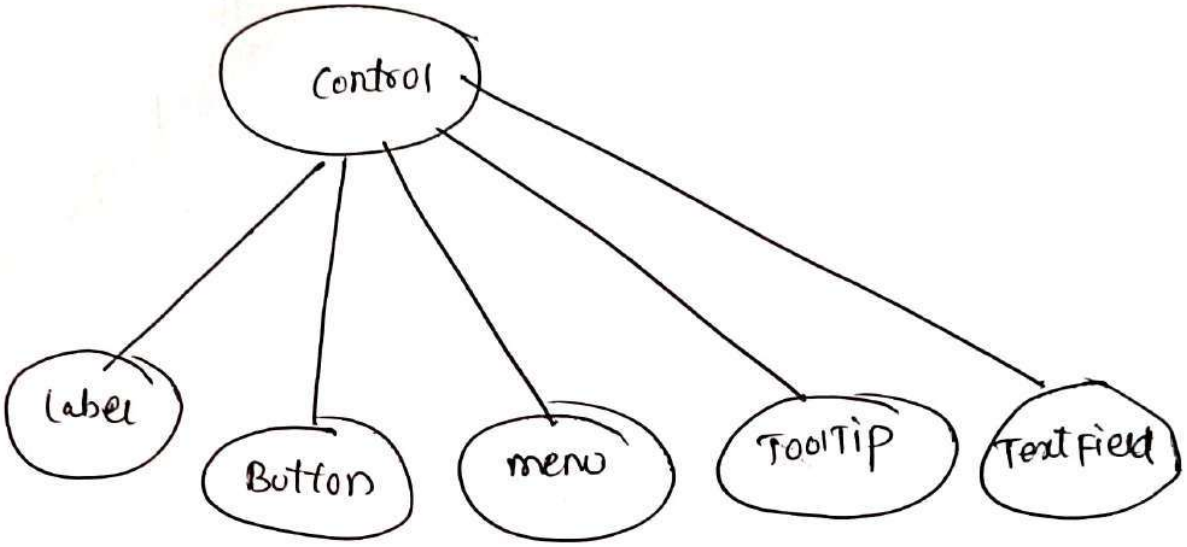
→ Libraries such as
  - Advanced windowing Toolkit
  - and
  - Swing

→ JavaFX, these Java programmers can now develop GUI applications effectively with rich content.

JavaFX Basic UI Controls

→ JavaFX provides a variety of UI controls that allow a smooth interaction between users & application.

These control are listed below



(1) Label

→ It is a component for displaying text.

(2) Button

→ It is a class used for creating buttons

(3) Menu

→ Contains a list of commands or options

(4) ToolTip

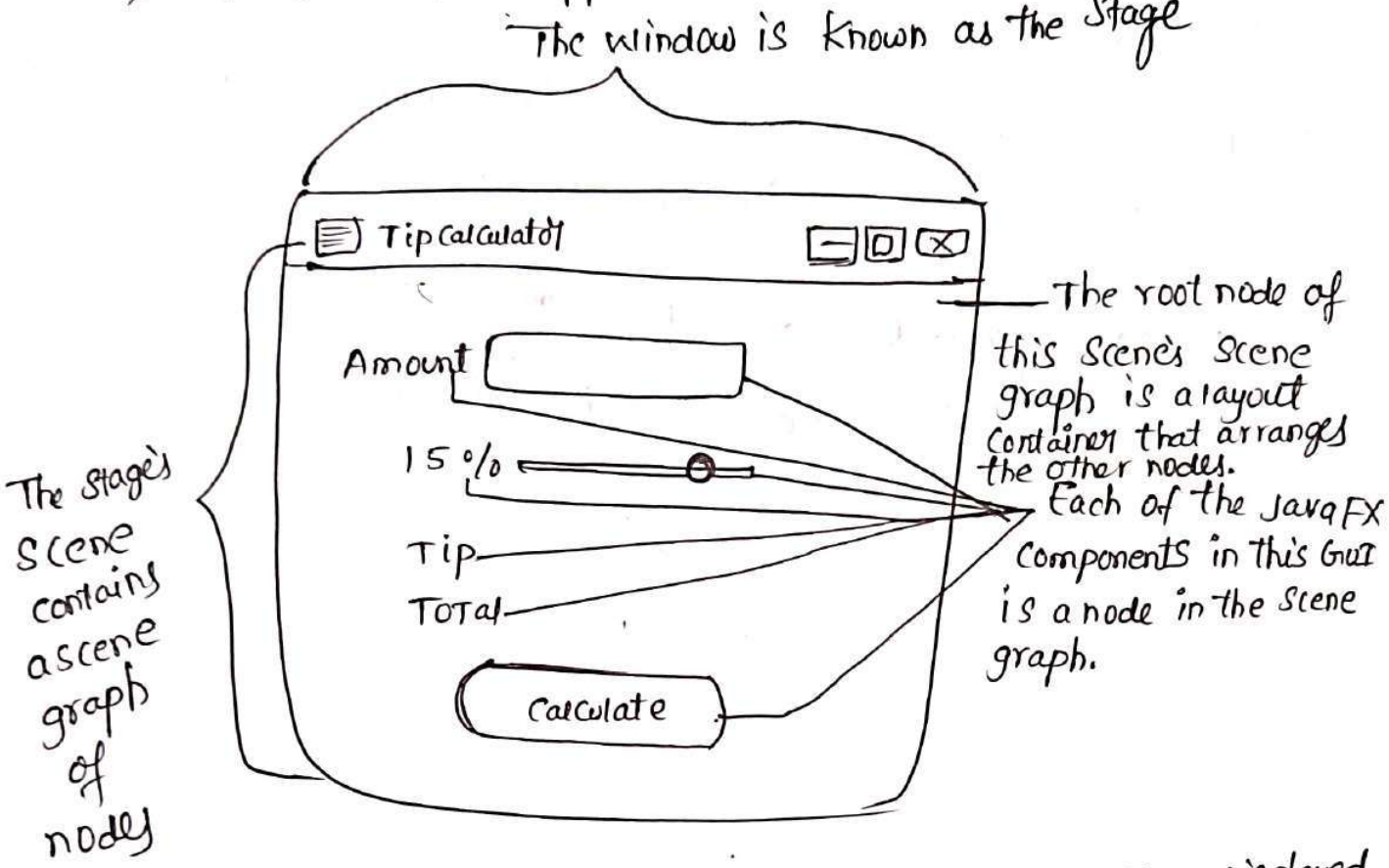
→ A pop-up window that displays some additional information about other UI elements

(5) TextField

→ Accepts & displays user input.

* Java FX App window Structure:

→ A Java FX app window consists of Several parts



→ The window in which a JavaFX app's GUI is displayed is known as the Stage and is an instance of class Stage (package javafx.stage).

→ The Stage contains one active Scene that defines the GUI as a Scene graph - a tree data Structure of an app's visual elements Such as GUI Controls

- ↳ shapes
- ↳ imagy
- ↳ video
- ↳ text & more

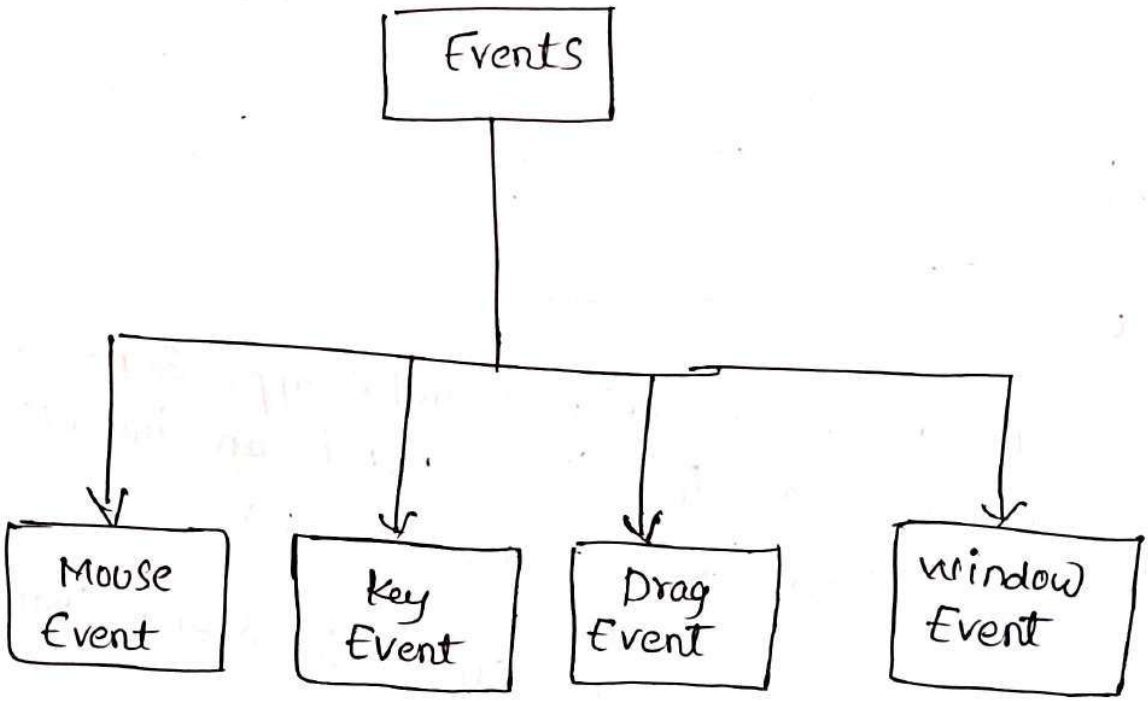
the Scene is an instance of class Scene (package javafx.scene.---)

* Event Handling :

→ JavaFX provides support to handle a wide variety of events.

→ The class named Event of the package javafx.event is the base class for an event.

→ JavaFX provides a wide variety of events. Some of them are listed below.



Mouse Event

→ This is an i/p event that occurs when a mouse is clicked.

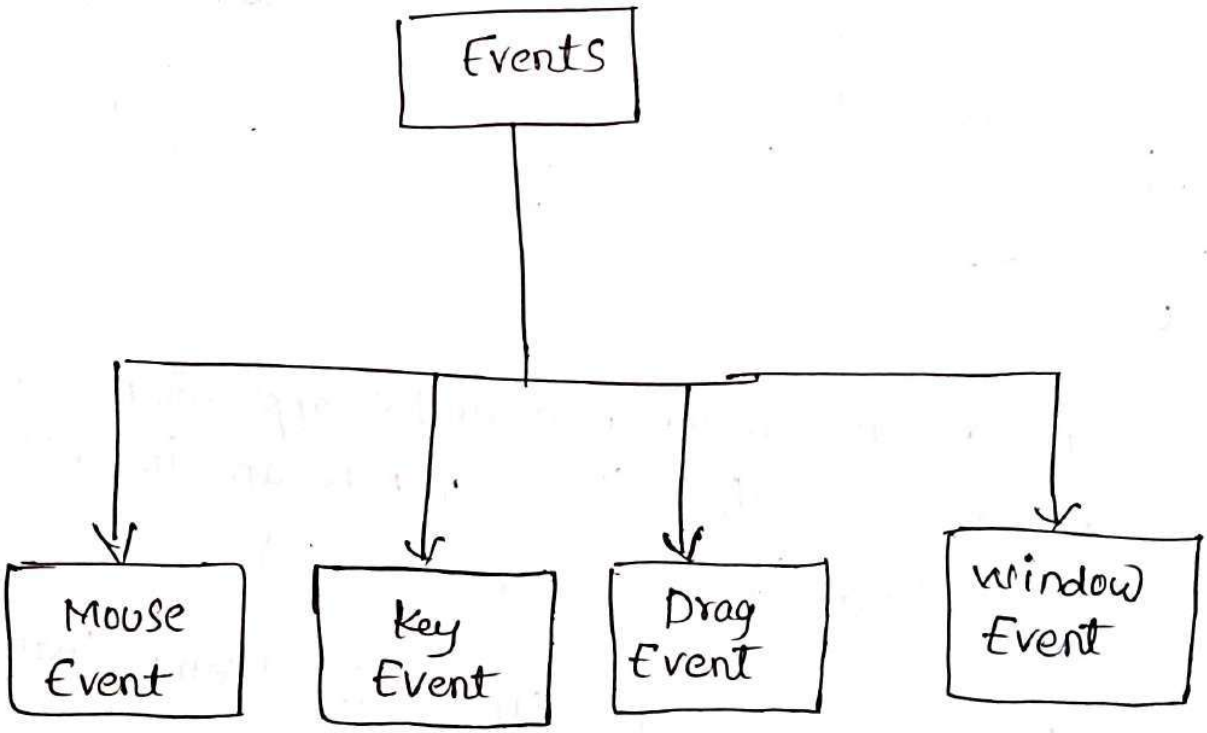
→ It is represented by the class named MouseEvent.

* Event Handling :

→ JavaFX provides support to handle a wide variety of events.

→ The class named Event of the package javafx.event is the base class for an event.

→ JavaFX provides a wide variety of events. Some of them are listed below.



Mouse Event

→ This is an i/p event that occurs when a mouse is clicked.

→ It is represented by the class named MouseEvent.

→ It includes action like
- ↳ mouse clicked
- ↳ mouse pressed
- ↳ mouse released
- ↳ mouse moved
- ↳ mouse entered target
- ↳ mouse exited target.

(2) Key Event:

→ This is an input event that indicates the Key Stroke occurred on a node.

→ It is represented by the class named KeyEvent.

→ This Event includes actions like
- ↳ Key pressed
- ↳ Key released

&

- ↳ Key typed.

(3) Drag Event:

→ This is an input Event which occurs when mouse is dragged.

→ It is represented by the class named DragEvent.

→ It includes actions like
- ↳ drag entered
- ↳ drag dropped
- ↳ drag over

(4) Window Event

→ This is an Event related to window showing / hiding actions.

→ It is represented by the class named WindowEvent

→ It includes actions like
  - ↳ window hiding
  - ↳ window Show

Event Handling

→ Event Handling is the mechanism that controls the event. & decides what should happen, if an event occurs.

→ This mechanism has the code which is known as an event handler that is executed when an Event occurs.

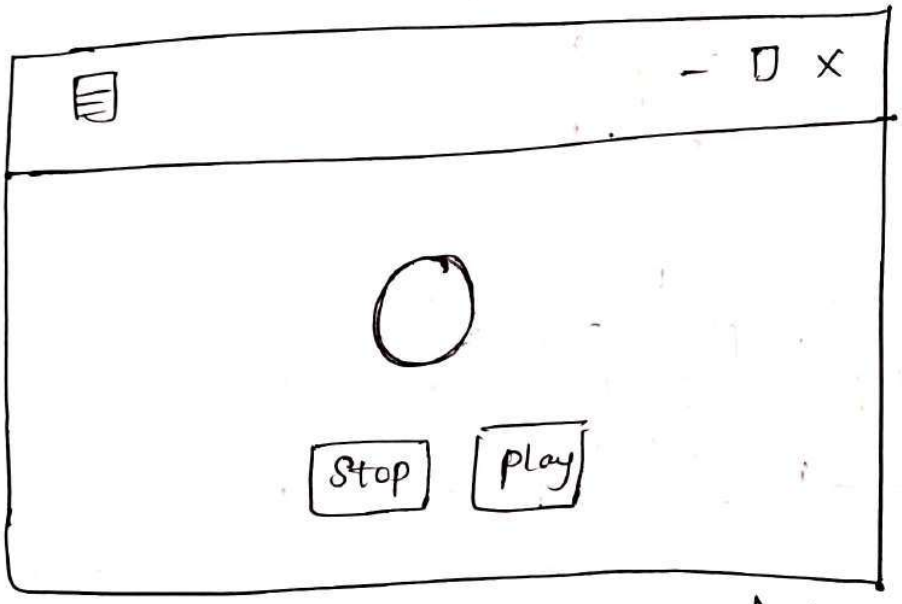
JavaFX provides handlers and filters to handle events.

Target – The node on which Event occured. A target can be a window.

Source – The Source from which the event is generated will be the Source of the event. mouse is the Source Event.

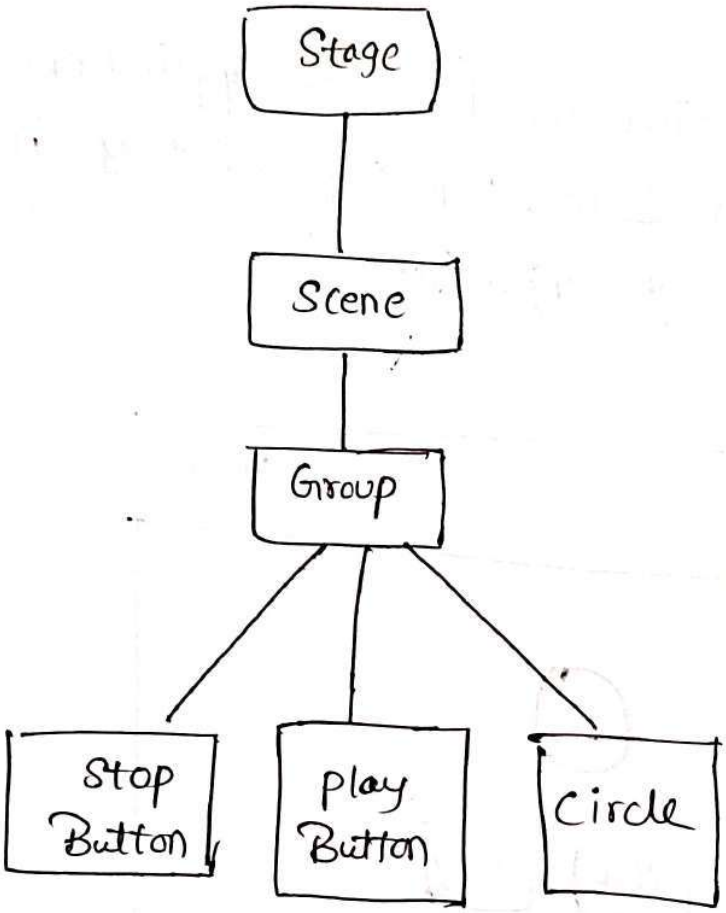
Type — Type of the occured event; in case of mouse event — mouse pressed, mouse released are type of events.

Assume that we have an application which has a circle, Stop & play Buttons inserted using a group of object as follows



If you click on the play button, the Source will be the play button and the type of the Event generated is the mouse click.

Phases of Event Handling in JavaFX:

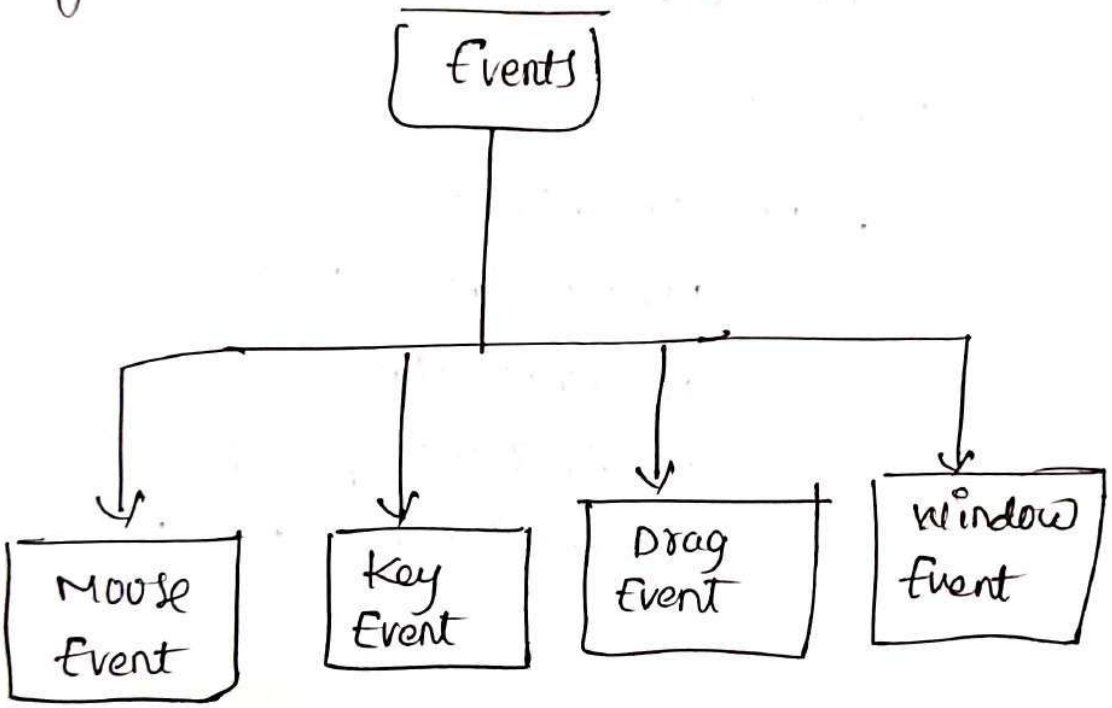


* Mouse Event:

→ JavaFX provides Support to handle a wide variety of Events.

→ The class named Event of the package java.fx.event is the base class for an event.

→ JavaFX provides a wide variety of Events some of them are listed below.



Mouse Event:

→ This is an input Event that occurs when a mouse is clicked.

→ It is represented by the class named mouse Event.

→ It includes action like

- mouse clicked
- mouse pressed
- mouse released
- mouse moved
- mouse entered target
- mouse exited target

Ex

javafx.scene.input

Class MouseEvent

- java.lang.Object
  - java.util.EventObject
  - javafx.event.Event
  - javafx.scene.input.InputEvent
  - javafx.scene.input.MouseEvent

* Display text and image:

→ A JavaFx application can consists of a lot of elements including all kinds of media like images, videos and all dimensional shapes, text etc.

→ All these elements are represented by nodes on a Javafx Scene graph.

→ But you can also create a Text element in javaFX applications. The text element is represented by a separate node & it can be altered with respect to its font, size, color and some other properties.

JavaFx Text Node

→ The textnode in JavaFX is represented by the class named Text, which belongs to the package javafx.scene.text.

→ This class contains several properties of the text like ↳ font
↳ alignment
↳ line spacing
↳ text etc

Creating Text Node:

→ Class Text of the package javafx.scene.text represents the textnode in JavaFx.

```
Text text = new Text();
```

→ After instantiating the Text class, you need to set value of this property using the setText() method as follows:

```
String text = "Hello how are you";
Text.setText(text);
```

→ you can set the position of the text by specifying the values of the properties x & y using their respective setter methods namely setX() and setY()

```
text.setX(50);
text.setY(50);
```

Example:

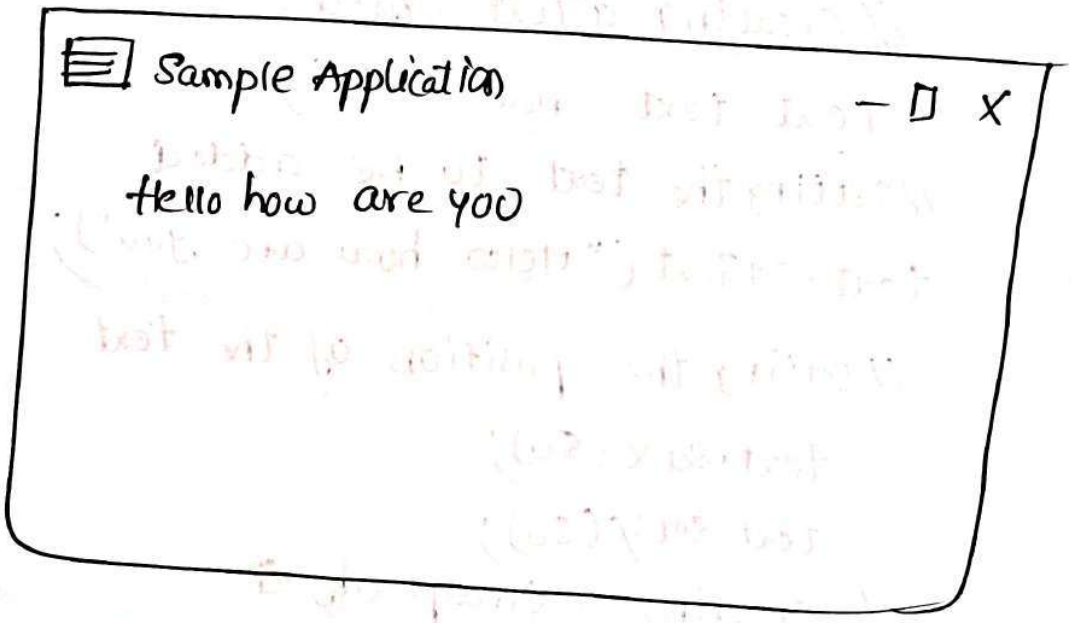
/* The following program is an example demonstrating how to create a text node in JavaFX */

```
import javafx.application.Application;
import javafx.scene.Group;
import javafx.stage.Stage;
import javafx.scene.text.Text;

public class TextExample extends Application
    {
    public void start(Stage stage)
      // creating a Text object
       Text text = new Text();
      //setting the text to be added
      text.setText("Hello how are you");
      // setting the position of the text
         text.setX(50);
         text.setY(50);
       // creating a Group object
         Group root = new Group(Text);
       // creating a Scene object
         Scene scene = new Scene(root, 600,300);
       // setting title to the Stage
          stage.setTitle("Sample Application");
```

```
//Adding Scene to the Stage
    Stage.setScene(Scene);
//Displaying the contents of the Stage
    Stage.show();
  }
psvm (String args[])
  {
    launch(args);
  }
}
```

Output



Display image in JavaFX

→ The javafx.scene.Image class is used to load an image into a JavaFX application.

→ This supports
- BMP
- GIF
- JPEG
- & PNG formats

→ JavaFX provides a class named javafx.scene.image.Image view is a node that is used to display, the loaded image.

→ To display an Image in JavaFX

(1) Create a FileInputStream representing the image you want to load.

(2) Instantiate the Image class bypassing the input Stream object created above, as a parameter to its Constructor.

(3) Instantiate the ImageView class.

(4) Set the Image to it by passing above the Image object as a parameter to the setImage() method.

(5) Set the required properties of the image view the respective setter methods.

(6) Add the Image view node to the group objec

Ex

```
import javaFX.stage.Stage;
import javaFX.io.FileInputStream;
import javaFX.io.IoException;
import javaFX.scene.Image.Imageview;
import javaFX.scene.Image;
import javaFX.Scene.Scene;
import javaFX.scene.Group;
public class ImageviewExampleExample extends Application
{
    public void Start(Stage stage) throws IoException
    { // creating the image object
InputStream Stream = new FileInputStream("D:\\Images\\apple.jpg");
    Image image = new Image(Stream);
    //creating the image view
  Imageview imageview = new Imageview();
  // setting the image to the image view
    imageview.SetImage(image);
  // setting the image view parametey
      imageview.setX(10);
      imageview.setY(10);
  // setting the scene object
  Group root = new Group(imageview);
  Stage.Scene scene = new Scene(root, 595, 370);
```

```
Stage.setTitle("Displaying Image");
Stage.setScene(Scene);
Stage.show();
}
psvm(String args[])
    {
launch(args);
    }
}
```

```
Stage.setTitle("Displaying Image");
Stage.setScene(Scene);
Stage.show();
}
psvm(String args[])
    {
    launch(args);
    }
}
```

* Java FX Scene Builder :

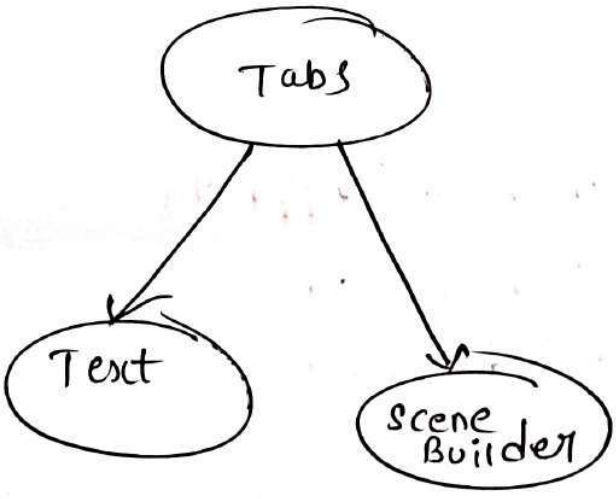
→ JavaFX Scene Builder is a visual layout tool that lets users quickly design JavaFX application User interfaces, without coding.

→ Scene Builder is free and opensource.

→ IntelliJ IDEA allows you to open .fxml files in JavaFX Scene Builder right from the IDE so that you can visually design JavaFX interfaces directly.

Open files in Scene Builder in the IDE:

→ When you open an .fxml file in the editor. there are two tabs in editing area



Text Tab:

→ Text tab is for developing the markup

Scene Builder:

→ Scene Builder tab is for editing the file in Scene Builder.

Configure the path to Scene Builder:

Step 1:

→ Download and install the latest version of Scene Builder.

Step2

→ press Cntr Alt S to open Settings and then Select language & JavaFX

Step 3

→ Click in the & path to Scene Builder ~~Dat~~ field.

Step4

→ Specify the path to the Scene Builder executable file on your Computer.

Path:

windows

C:\Users\UserName\AppData
\Local\Scene Builder
\Scene Builder.exe

Step5

→ Apply changes & close the dialog.