1) Java Basic program :- Sample.java

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

o/p: Hello World

Compilation

javac Sample.java

Execution

java

Keyword (class public static)

S → capital (uppercase)

Compilation :- javac Sample.java.

Execution :- java Sample

↳ By default it is sample.class file.

output :- Hello World.

2) demo.java

```
class demo
{
    public static void main (String [] args)
    {
        System.out.println (78);
        System.out.println (1.6);
        System.out.println ('@');
        System.out.println (true);
        System.out.println ("Sanju");
    }
}
```

Compilation → javac demo.java

execution → java demo

Output :- 78
1.6
@
true
sanju

Primitive data types → predefined

Integer — Byte — 1 byte
short — 2 byte
int — 4 byte
dong — 4 byte

Decimal — float — 4 byte
double — 8 byte

char — char

Boolean — Boolean

Memory
|
Store
|
data
pgm operation } use

Non primitive data type
↓ String.
user defined

05/02/19

DataType

DataType :— Data-type specifies three things
(i) allocate memory block.
(ii) Specifies Type of data
(iii) Specifies memory size.

There are two different data types.
(i) primitive Data Types (predefined)
(ii) Non primitive Data Types (user defined)

(i) primitive Data Types :—

There are 8 types
1) byte → 1 byte
2) short → 2 byte
3) int → 4 byte
4) long → 8 byte
5) float → 4 byte
6) double → 8 byte
7) char → 2 bytes
8) boolean →

(ii) Non primitive Data types :-

String → To handle group of characters

Variable :-

→ It is a memory block, which has 4 characteristics.

(i) Variable name
(ii) Variable type
(iii) Variable size
(iv) Variable data

→ In order to use a variable we have to follow 2 steps.

Step 1 : Variable declaration (Reserve Memory)

* Syntax : datatype variable name ;

Step 2 :- Variable Initialization (Storing Data)

* Syntax : variable name = data ;
L ↷ R

```
int i;
i = 23;
```
i | 23

```
double j;
j = 2.3
```
j | 2.3

```
char k;
k = '@';
```
k | '@'

```
boolean x;
x = true;
```
x | true

```
String y;
y = "Sanju";
```
y | "Sanju"

Note: We can declare and initialize a variable in a single statement

Syntax : datatype variablename = data;

Ex:
```
int i = 23;
double j = 2.3;
char k = '@';
boolean x = true;
String y = "Sanju";
```

→ Write a program to store all types of data and print it.

```
class demo
{
    public static void main(String[] args)
    {
        int i = 10;
        double j = 2.3;
        char k = '@';
        boolean x = true;
        String s = "sanju";
        System.out.println(i);
        System.out.println(j);
        System.out.println(k);
        System.out.println(x);
        System.out.println(s);
    }
}
```

06/02/19

'+' operator :- It has two properties.

(i) Addition

(ii) Concat. → joining.

→ If inputs are of number type or character type, then + operator behaves like addition operator.

Note :- In case of a character, it gets converted to ASCII Number.

Ex:

1) 5 + 6 → 11

2) 1.2 + 2.7 → 3.9

3) 1.2 + 6 → 7.2

4) 5 + 'A' → 70 (65)

5) 'A' + 'B' → 131

→ If one of the input is string type, then '+' operator behaves like concat operator.

→ After joining entire result becomes another string.

Ex:

1) "abc" + 5 → "abc5"

2) 6 + "abc" → "6abc"

3) true + "abc" → "trueabc"

4) '@' + "pqr" → "@pqr"

5) "xyz" + 5.1 → "xyz5.1"

6) "abc" + "xyz" → "abcxyz"

Interview Questions

1) "abc" + 10 + 10 → "abc1010"

"abc" + 10 + 10

"abc10" + 10

"abc1010"

2) 10 + 10 + "abc" → "20abc"

10 + 10 + "abc"

20 + "abc" → "20abc"

3) "abc" + (10 + 10)

"abc" + 20 → "abc20".

Note:- (1) int k = 78;

System.out.println(k); → 78

System.out.println("k is " + k); → k is 78.

O/P

(2) double j = 2.3;

System.out.println(j); → 2.3

System.out.println("j is " + j); → j is 2.3

O/P

(3) String x = "raja";

System.out.println(x); → raja

System.out.println("My name is " + x); → My name is raja

O/P

(4)

* Write a logic to store yourname and your age. and print with meaningful mess message.

```
String s = "Sandhya";
int i = 23;
System.out.println("My name is " + s +
                   "and My age is " + i);
```

→ O/P: My name is Sandhya and My age is 23.

* Write a logic to store your name and your Mothers name and output should be in the below manner

Ex: Raja S/O Son of Maharani

```
String s = "Sandhya";
String d = "Dhanalakshmi";
System.out.println(s + "daughter of " + d);
```

→ O/P: Sandhya daughter of Dhanalakshmi

* Write a logic to store your name, age, height and print the output in below manner.

Ex: Raja 23 years old with 5.6 feet height.

```
String s = "Raja";
int i = 23;
double d = 5.6;
System.out.println(s + i + "years old with" + d +
                   "feet height");
```

O/P:- Raja 23 years old with

→

```
int a = 5;
int b = 6;
int c = a+b;
System.out.println("Addition of" + a + "and" + b +
                   "is" + c);
```

07/02

Keywords and Identifiers:-

Keywords:-

(i) These words are inbuilt words with predefined special meaning

(ii) All the keywords must be used in lowercase.

(iii) Keywords should not be used as identifiers.

(iv) List of keywords:-

1) class
2) public
3) static
4) void

5) byte

6) short

7) int

8) long

9) float

10) double

11) char

12) boolean

13) if

14) else

15) else if

16) switch

17) case

18) break

19) return

20) new

21) this

22) extends

23) super (args / No args)
↓
call to super.

24) super
↓
super statement

25) final

26) abstract

27) interface

28) implements

29) instance of (keyword & returns boolean)

30) private

31) protected

32) import

33) package

Identifiers :-

(i) These words are given by programmer to identify,

1) class → class name

2) variable → variable name

3) Method → Method name

4) package → package name

5) project → project name

Rules of Identifiers :-

(i) Identifier can be a combination of below list.

A-Z
a-z
0-9
$, _ (special)

Ex: demo → valid
DEMO → valid
demos → valid
Demo → valid
demo_pgm → valid

demo@pgm → Invalid
demo$ pgm → valid
demo pgm → Invalid

(ii) An Identifier can start with alphabet & special character but should not start with digit/number.

RUN ✓
run ✓
RUN7 ✓

7RUN (X) Invalid
run ✓
$run ✓

R7un ✓

_ underscore

Standard naming format (or) Convention for Identifiers :-

1) class naming format :-

→ Every word first letter should be capitalized.

Ex:1 -
```
class Demo
{
---
}
```

Ex:2 :-
```
class SampleProgram
{
---
}
```

2) Variable naming format :—

→ If variable name consists of only one word, entire word should be written in lower case.

Ex: int id;

double marks;

→ If variable contains multiple words. first word should be in lower case from 2nd word onwards first letter should be capital

Ex: int empId;

int studentYearOfPassout;

3) Method naming format :—

→ If method name consists of only one word, entire word should be in lower case.

→ If method name consists of multiple words, first word should be in lower case, from second word onwards first letter should be capital.

Ex: 1 ⇒ display()

Ex 2 :— empInfo();

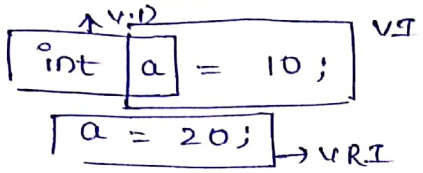
4) package naming format :—

5) project naming format :—

Summary :— By following above standard naming formats we can improve readability of the program.

Note :— variable reinitialization :—

Ex-1 :



a | 10 | 20

Ex-2 :

```
double j = 2.5;
j = 5.5;
```

j | 2.5 | 5.5

V.D → variable declaration

VI → variable Initialization

VRI → variable reinitialization.

Note :—

→ If variable types are matching we can copy value from one variable to another variable.

```
int a = 25;
int b;
b = a;
System.out.println(a);
System.out.println(b);
```

O/p: 25
25

Combination of operators :—

+=, -=, *=, /=, %=

Ex: 1 →

```
int t = 36
S.O.pln(t);
t = t+4; / t+=4;
S.O.pln(t)
```

t | ~~36~~ | 40

Ex 2:
```
int a = 79;
S.O.pln(a);
a = a-2; / a -= 2;
S.O.pln(a);
```

a | 79 | 77

Ex: 3:
```
int k = 35
S.O.pln(k);
k = k * 3; / k *= 3;
S.O.pln(k);
```

k | 35 | 105

Arthmatic operators :-

+

-

*

/ → This operator after divison returns quotient as output

% → This operator after divison returns remainder. as output

Ex:
```
int i = 10;
int j = 2;
S.O.pln(i/j);
S.O.pln(i%j);
```

i | 10

j | 2

Output :-

5 → quotient

0 → remainder

08/02

* Data type gives memory block.

Increment and Decrement Operator (++ & --)

Ex:
```
int a = 10;
int b = a++;
S.O.pln(a);  → 11
S.O.pln(b);  → 10
```

Ex:
```
int x = 46;
int y = ++x;
S.O.pln(x);  → 47
S.O.pln(y);  → 47
```

In other words
- a++ → give the value to a & then incremented
- ++a → first incremented, then the incremented value given to a.

Ex:
```
int m = 92;
int n = --m;
S.O.pln(m);  → 91
S.O.pln(n);  → 91
```
m [92]

Ex:
```
int i = 89;
int j = i--;
S.O.pln(i);  → 88
S.O.pln(j);  → 89
```

Ex:
```
int a = 72
S.O.pln(++a + a++)
```
++a → 73 → a [73]
a++ → 74
73 + 73 = 146
S.O.pln(a); → 74

a = 72
a = 73
a = 74
① → increment 1
++a → 73
② assign a = 73

a++ ② increment +1 (73+1) = 74
① 73
a = 74

Ex:
```
int a = 45;
S.O.pln(a-- + ++a);
S.O.pln(a);
```
45 + 45 = 90
→ 45

a-- ① → 45 → 44
++a → 45
44 + 1 = 45
a = 45

Rough work

a = 10
b = 10
c = 12
d = 11 (= a)
e = 11 a--

c = ++a (12)
a = 11
11
10 a
12

++a (10 11 12)
11 + 11 = 22
12 + 12 = 24
13 + 11 - 11
12
a = 10

Ex: int a = 92;

S.O.pln (--a + a++);

S.O.pln (a); → 92

--a → 92-1 = 91, a = 91

a++ → 91 + 1 = 92

Ex: int a = 67;

S.O.pln (++a + --a - a--);

S.O.pln (a); → 68 + 67 - 67

66

68

++a → 67+1 = 68

--a → 68-1 = 67

a-- → 67-1 = 66

Ex: int a = 48;

S.O.pln (a-- + ++a + a-- + ++a);

96 ← 48 + 48 + 48 + 48

S.O.pln (a); → 48

a: 48 47 48

a-- → 48 47

++a → 47+1 = 48, a = 48

a-- → 48 → 47

++a → 48

a = 92 93

b 16 15

Ex: int a = 92;

int b = 16;

S.O.pln (a++ - b-- + --a);

S.O.pln (a); → 92 - 16 + 92 ⇒ 168

S.O.pln (b) → 15

a++ → 92+1 = 93

b-- → 16 → 15

--a → 93-1 = 92

92 - 16 = 168

Ex: int a = 75;

75 ← S.O.pln (a); → a = 75

75 ← S.O.pln (a++); → a++ → 75, 75+1 = 76

77 ← S.O.pln (++a); → ++a → 76+1 = 77

76 ← S.O.ph (--a); → --a → 77-1 = 76

75 ← S.O.pln (--a); → --a → 76-1 = 75

75 ← S.O.pln (a--); → a-- → 75, 75-1 = 74

74 ← S.O.ph (a); → a = 74

74 ← S.O.ph (a++); → a++ = 74, 74+1 = 75

75 ← S.O.ph (a); → = 75

75 75 76 77 76 75 75 74 74 75

a 74

Ex: int a = 11; a|11

a = a++;

sopln(a); → 11

a++

↓

11 11+1=12

In this statement a++ operation will first assign the value and then incremented but In java single statement can assign the value only once.

11/02 :-

1. pre increment & decrement says,
   (i) first do the job (increment/decrement).
   (ii) later give the value.
2. post increment & decrement says,
   (i) first give the value.
   (ii) later do the job (increment/decrement).

Relational operator :-

< → less than

> → greater than

<= → less than or equal to

>= → greater than or equal to

== → Comparison operator.

!= → Not equals to

input → number

output → boolean

Ex: 6 > 4 → True

5 < 4 → false

5 == 5 → True

6 != 6 → false

Logical operators :-

&& → AND

|| → OR

! → NOT

input → boolean

output → boolean.

AND

Cond1	Cond2	O/P
T	T	T
T	f	f
F	T	f
F	F	F

OR

Cond1	Cond2	O/p
T	T	T
T	F	T
F	T	T
F	F	F

NOT

Cond	O/p
T	F
F	T.

Ex: (5<4) && (6>4)

f T

F

Ex: (5<4) || (6>4)

f T

T

Ex: !(7>=7)

T

False.

Note :- According to standard practise, when logical operators are use, conditions must be provided with in the brackets. By doing so we improve readability.

Control Statements :-

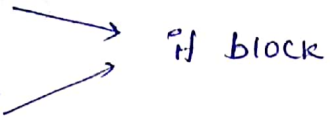
1. In order to control flow of the program we use these statements.
2. There are two types of control statements.
   (i) Conditional statements / decision making statement
   (ii) Looping statements / Iteration Statement

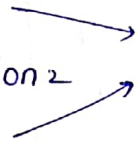
(i) Conditional Statements / decision making statement :-

if else logic :-

Syntax :

```
if (cond)
{
    ----
    ----        operation 1
    ----
    ----
}
else
{
    ----
    ----        operation 2
    ----
    ----
}
```





1. If condition is true if block operation gets executed.
2. if condition is false else block operation gets executed.

Write a program to check whether given age is elible for election voting.

Note 1 :- provide the appropriate classname based on the program.

Note 2 : provide the appropriate variable name based on the data.

→ By following above standard practise we can improve programs readability.

```
class ElectionEligibility
{
    public static void main (String[] args)
    {
        int age = 23 ;
        if (age >= 18)
        {
            S.O.pln ("Eligible for election voting");
        }
        else
        {
            S.O.pln ("Not Eligible for Election voting);
        }
    }
}
```

O/p : Eligible for Election voting.

Write a program to check whether given number is divisible by 4.

divisor) dividend (quotient (/)

remainder (%)

```
class DivisibleBy4
{
    public static void main (String[] args)
    {
        int n = 32;
        if (n%4 == 0)
        {
         S.O.println ("Given number is divisible by 4");
        }
        else
        {
        S.O.println ("Given number is not divisible by 4");
        }
    }
}
```

Write a program to check whether given number is even & odd.

```
class EvenOdd
{
    public static void main (String[] args)
    {
        int n = 8;
        if (n%2 == 0)
        {
          S.O.println ("Given number is even number");
        }
        else
        {
          S.O.println ("Given number is odd number");
        }
    }
}
```

Write a logic to check whether given age is eligible for to take up IAS Exam.

age criteria (25 to 36)

```
Class IasExam
{
    public static void main(String[] args)
    {
        int age = 23;
        if((age >= 25) && (age <= 36))
        {
          System.out.println("Eligible for to takeup IAS Exam");
        }
        else
        {
          System.out.println("NOt Eligible for to takeup IAS Exam");
        }
    }
}
```

o/p : NOt Eligible for to takeup IAS Exam.

12/02/19

logic :-

~~else if~~ statement :-

else-if logic :-

Syntax :-

```
if (cond 1)
{
  ------
  ------
}
else if (cond 2)
{
  ------
  ------
}
else
{
  ------
  ------
}
```

if block → Oper 1

elseif block → Op2

optional ← else block → op 3

Note 1 :- We can have multiple else if conditions.

Note 2 :- We can either end with else if condition or else condn

Prog: Write a program to print students rank based on students grade.

Refer below table.

Grade	A	B	C	D	E	F
Rank	1st	2nd	3rd	1st class	2nd class	fail

```
Class Rank
{
  public static void main (String[] args)
  {
    char grade = '@';
    if ( grade == 'A')
    {
      S.O.pln ("1st Rank");
    }
    else if (grade == 'B');
    {
      S.O.pln ("2nd Rank");
    }
    else if (grade == 'C')
    {
      S.O.pln ("3rd Rank");
    }
    else if (grade == 'D')
    {
      S.O.pln ("1st class");
    }
    else if (grade == 'E')
    {
      S.O.pln ("2nd class");
    }
    else if (grade == 'F')
    {
      S.O.pln ("fail");
    }
    else
    {
      S.O.pln ("Invalid grade");
    }
  }
}
```

O/p: Invalid grade

(Not write)

```
else
{
  S.O.pln ("fail");
}
```

Write a logic to modify above program in the below manner refer the table.

Grade	A & a	B & b	C & c	D & d	E & e	F & f
Rank	1st Rank	2nd Rank	3rd Rank	1st class	2nd class	failure.

```
class Rank
{
  public static void main (String[] args)
  {
      char grade = 'D';
      if ((grade =='A') || (grade=='a'))
      {
       S.O.pln (" 1st Rank");
      }
     else if ((grade=='B') || (grade = = 'b'))
       {
        S.O.pln (" 2nd Rank");
       }
     else if ((grade = = 'c') || (grade = = 'c'))
       {
        S.O.pln ("3rd Rank");
       }
     else if ((grade = = 'D') || (grade = = 'd'))
       {
       S.O.pln (" 1st class");
       }
     else if ((grade = = 'E') || (grade = = 'e'))
       {
        S.O.pln (" 2nd class");
       }
     else if ((grade = = 'F') || (grade = = 'f'))
       {
        S.O.pln (" failure");
       }
      else
       {
        S.O.pln (" Invalid grade");
       }
  }
}
```

O/P = 1st class

Write a program to check whether lovers are eligible for marriage.

```
class Marriage
{
  public static void main (String[] args)
  {
      int age1 =     ;
      int age2 =     ;
     if ( (age1 >= 18) && (age2 >= 21))
     {
       S.o.pln ("Lovers are eligible for marriage");
     }
     else if ((age1 >= 18) && (age2 <= 20))
     {
       S.o.pln ("Lovers are not eligible for marriage");
     }
     else if ((age1 <= 17) && (age2 >= 21))
     {
       SO.pln ("Lovers are not eligible for marriage");
     }
     else if ((age <= 17) && (age <= 20))
     {
       S.o.pln ("Lovers are not eligible for marriage");
     }
  }
}
```

T T
T F
F T
F F

13/02

Switch Statement :-

(i) if there are multiple equality test condition to perform, we should use switch statement.

(1) If expression is having matching case, start the execution from that case and continue the execution until break statement or until end of switch statement.

(2) If expression is not having matching case, start the execution from default and continue the execution until break statement (or) until end of switch statement.

(3) Switch can have multiple cases

(4) case value should be unique.

(5) Switch can have only one default statement

(6) Expression type can be anything (integer (or) decimal (or) character (or) string (or) Boolean).

(7) Expression type and value type should be same.

(8) Break statement is optional.

(9) default statement is also optional.

(10) case value can be ordered (or) unordered.

(11) default statement can appear anywhere with in the switch statement.

Syntax :-

```
switch (exp)
{
    case value 1 :
                  ---
                  ---
                  ---
                  break;
    case value 2 : ----
                  ----
                  ----
                  break;
    case value 3 : ----
                  ----
                  ----
                  break;
    .
    .
    .
    default : ----
              ----
              ----
              break;
}
```

Modify Students grade and rank program using switch statement

Grade	A	B	C	D	E	F
Rank	1st Rank	2nd Rank	3rd Rank	1st class	2nd class	fail

```
class ModifyGradeRank.
{
  public static void main (String[] args)
  {
     char grade = 'C';
     switch (grade)
     {
       case 'A' : S.O.pln ("1st Rank");
                  break;
       case 'B' : S.O.pln ("2nd Rank");
                  break;
       case 'C' : S.O.pln ("3rd Rank");
                  break;
       case 'D' : S.O.pln ("1st class");
                  break;
       case 'E' : S.O.pln ("2nd class");
                  break;
       case 'F' : S.O.pln ("fail");
                  break;
       default : SO.pln ("Invalid");
     }
  }
}
```

output:-

3rd Rank.

Write a program to print dayname based on daynumber. refer below table

day number	0	1	2	3	4	5	6	7
dayname	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday

```
class Dayname
{
  public static void main(String[] args)
  {
     int dayNum = 2;
     switch (dayNum)
     {
       case 0: S.O.pln("Sunday"); break;
       case 1: S.O.pln("Monday"); break;
       case 2: S.O.pln("Tuesday"); break;
       case 3: S.O.pln("Wednesday"); break;
       case 4: S.O.pln("Thursday"); break;
       case 5: S.O.pln("Friday"); break;
       case 6: S.O.pln("Saturday"); break;
       case 7: S.O.pln("Sunday"); break;
       default: S.O.pln("Invalid");
     }
  }
}
```

Output:-

Tuesday.

* In the above program there is a repeatative code. We can avoided by following below steps.

Step 1:- Keep the repeatative code together in one place.

Step 2:- Remove the repeatative code from all the cases except last case.

```
class Dayname
{
  public static void main (String[] args)
  {
     int dayNum = 1;
     Switch (dayNum)
     {
       case 7 :
       case 0 : S.O.pln ("Sunday"); break;
       case 1 : S.O.pln ("Monday"); break;
       case 2 : S.O.pln ("Tuesday"); break;
       case 3 : S.O.pln ("Wednesday"); break;
       case 4 : S.O.pln ("Thursday"); break;
       case 5 : S.O.pln ("Friday"); break;
       case 6 : S.O.pln ("Saturday"); break;
     } default : S.O.pln ("Invalid");
  }
}
```

output:—

Monday.

Write a program to print students Rank based on students grade refer below table.

Grade	A & a	B & b	C & c	D & d	E & e	F & f
Rank.	1st	2nd	3rd	1st class	2nd class	fail

any other character Invalid grade.

```
class Rank
{
  public static void main (String[] args)
  {
     char grade = 'a';
     Switch (grade)
     {
```

```
case 'A' :
case 'a' : S.o.pln ("1st Rank"); break;
case 'B' :
case 'b' : S.o.pln ("2nd Rank"); break;
case 'C' :
case 'c' : S.o.pln ("3rd Rank"); break;
case 'D' :
case 'd' : S.o.pln ("1st class"); break;
case 'E' :
case 'e' : S.o.pln ("2nd class"); break;
case 'P' :
case 'f' : S.o.pln ("fail"); break;
default : S.o.pln ("Invalid grade");
}
}
}
```

Output :-

1st Rank.

Assignment :-

1) Write a program to print season name based on month number. Refer the below table.

3, 4, 5	6, 7, 8.	9, 10, 11	12, 1, 2	Month
Summer	Rainy	Spring	Winter	season

2) Write a program to print current day name and remaining future dayname till the end of the week based on day number. Refer below table.

day number	1	2	3	4	5	6	7
day name	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday

any other number invalid day number.

Assignment :-

1.
```
class Season
{
    public static void main (String [] args)
    {
        int month = 6 ;
        Switch (month)
        {
        case 12 :
        case 1 :
        case 2 : S.o.pln (" Winter Season");
                 break;
        Case 3 :
        case 4 :
        case 5 : S.O.pln (" Summer Season");
                 break;
        Case 6 :
        case 7 :
        case 8 : SO.pln (" Rainy Season");
                 break;
        case 9 :
        case 10 :
        case 11 : S.O.pln (" Spring Season");
                  break;
        default : S.O.pln (" Invalid ");
        }
    }
}
```

```
Case 3 :
    4
    5

Case 6
    7
    8

    9
    10
    11

    12
    1
    2
```

output :-

Rainy Season

2. Class CurrentDayNameAndFutureWeekDayName

```
{
  public static void main (String [] args)
  {
    int dayNum = 2;
    Switch (dayNum)
    {
      case 1 : S.O.pln (" Monday ");
      case 2 : S.O.pln (" Tuesday ");
      case 3 : S.O.pln (" Wednesday ");
      case 4 : S.O.pln (" Thursday ");
      case 5 : S.O.pln (" Friday ");
      case 6 : S.O.pln (" Saturday ");
      case 7 : S.O.pln (" Sunday "); break;
      default : S.O.pln (" Invalid ");
    }
  }
}
```

Output :-

Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday.

14/02

Looping Statement :-

1. We go for looping statement inorder to repeatatively execute the given task.

for loop :-

Syntax :

```
for (arg1 ; arg2 ; arg3)
{
  ========   task to execute again and again
}
```

arg1 : Variable declaration and initialization

arg2 : Condition

arg3 : increment (or) decrement statement.

Note :-

Given task gets executed again and again and till the condition is false.

Write a logic to print Good Morning three times.

```
class printGoodMorning
{
    public static void main(String[] args)
    {
        for(int count = 1 ; count<=3 ; count++)
        {
            S.O.pln("Good Morning");
        }
    }
}
```

Increment

1<=3 (T) 2<=3 (T) 3<=3 (T) 4<=3 (F)

Good Morning.
Good Morning.
Good Morning.

(or)

```
class printGoodMorning
{
    public static void main(String[] args)
    {
        for(int count = 3 ; count >= 1 ; count--)
        {
            S.O.pln("Good Morning");
        }
    }
}
```

Decrement

Write a logic to print your name 10 times.

```
class printyourName
{
    public static void main(String[] args)
    {
        for(int count = 1 ; count <= 10 ; count++)
        {
            S.O.pln("K. Sandhya Sri");
        }
    }
}
```

Write a logic to print numbers from 1 to 10.

```
class printNumbers
{
   public static void main (String[] args)
   {
      for (int i = 1 ; i <=10 ; i++)
      {
         S.O.pln (i);
      }
   }
}
```

O/P:
1
2
3
4
5
6
7
8
9
10

Write a logic to print numbers from 30 to 45.

```
class printNumbers
{
   public static void main (String[] args)
   {
      for (int i = 30 ; i <= 45 ; i++)
      {
         S.O.pln (i);
      }
   }
}
```

O/P:
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45

Write a logic to print numbers from 10 to 1

```
class printNumbers
{
   public static void main (String[] args)
   {
      for (int i = 10 ; i >= 1 ; i--)
      {
         S.O.pln (i);
      }
   }
}
```

Output:
10
9
8
7
6
5
4
3
2
1

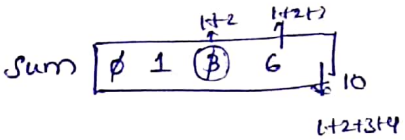
Write a logic to print 75 to 64

```
class print
{
  public static void main (String[] args)
  {
    for (int i = 75 ; i >= 64 ; i--){
    {
      S.O.pln (i);
    }
  }
}
```

o/p:

75
74
73
72
71
70
69
68
67
66
65
64

** Write a logic to find sum of first 10 Natural numbers.




```
class Sum
{
  public static void main (String[] args)
  {
    int sum = 0;
    for (int i = 1 ; i <= 10; i++)
    {
      sum = sum + i;
    }
    S.O.pln ("sum of first 10 natural numbers is : " + sum);
  }
}
```

Output:

Sum of first 10 natural numbers are 55.

Exp-1:

Experiment -1 : figure of the o/p:

```
for (int i = 1; i <= 10; i++)
{
  int sum = 0;
  sum = sum+i;
}
S.O.pln ("sum of first 10 natural numbers is : "+sum);
```

sum | 0 |

Output :- Error:

cannot find symbol

Memory block is not available.

Reason :-

Experiment – 2 : figure of the output

```
for (int i = 1 ; i <= 10 ; i++)
{
    int sum = 0;
    sum = sum + i;
    S.o.pln ("sum of 10 Natural numbers is " + sum);
}
```

Experiment – 3 : figure of the output

```
int sum = 0;
for (int i = 1 ; i <= 10 ; i++)
{
    sum = sum + i;
    S.o.pln ("sum of 10 natural numbers is " + sum);
}
```

Reason [Ex 1]:

Because variable sum is available only within the scope of for loop. Since we are using some sum variable outside the boundary/scope which is not available hence there is an error.

Reason for Ex-2 :-

For each loop sum variable memory block gets created and gets removed when control goes to next loop. Every for every loop sum variable gets created and value will be zero.

Reason for Ex-3 :

Every time the result gets printed.

Sum variable is alive for every loop till the exit of the main method.

Write a logic to find product of first 10 natural numbers.

```
class productOfNaturalNumbers
{
  public static void main (String[] args)
  {
     int prod = 1;
     for (int i = 1 ; i <= 10 ; i++)
     {
       prod = prod * i ;
     }
     S.O.pln (" product of first 10 natural numbers is : " + prod);
  }
}
```

1x2x3
1x2 1
prod | 1 2 6 |

Output:

product of first 10
natural number is 3628800

Write a logic to print all the even numbers and odd numbers present between 10 to 20.

```
class EvenNumAndOddNum
{
  public static void main (String[] args)
  {
     for (int i = 10 ; i <= 20 ; i++)
     {
       if ( i % 2 == 0)
       {
         S.O.pln ( i + "is even num " );
       }
       else
       {
         SO.pln ( i + " is odd num ");
       }
     }
  }
}
```

output:

10 is even numbe
11 is odd num
12 is even num
13 is odd num

Write a logic to print only even numbers present between 10 to 20

```
class printEvenNumbers
{
   public static void main(String[] args)
   {
      for(int i = 10; i<=20; i++)
      {
         if(i%2==0)
         {
            S.O.println("Below numbers are Even numbers");
            S.O.println(i);
         }
      }
   }
}
```

Output:

Below numbers are Even numbers

10
12
14
16
18
20

Assignment

1) Write a logic to print all the numbers which are divisible by 7 present between 19 to 36

2) Write a logic to find sum of all the numbers divisible by 5 present between 7 to 28.

3) Write a logic to find product of all the numbers divisible by 3 present between 15 to 35.

i 19 20 21

19<=36 (T) 20<=36

19%7 == 5 != 0

1.

```
class DivisibleNumbers
{
   public static void main(String[] args)
   {
      for(int i = 19; i<=36; i++)
      {
         if(i%7==0)
         {
            S.O.println("Below numbers are Divisible by 7");
            S.O.println(i);
         }
      }
   }
}
```

output:

below number is divisible by 7
21
below number is divisible by 7
28.
below number is divisible by 7
35.

2. Wrong.

```
class SumOfNumbersDivisibleBy5.
{
  public static void main(String[] args) {
  { int sum = 0;
    for (int i = 7; i <= 28; i++)
    {
      sum = sum + i;
      if (sum % 5 == 0)
      {
        S.O.pln(sum);
      }
    }
  }
}
```

Output :-
15 (Wrong)
40
75
115
180
210
255
330
385

3. Wrong

```
class productOfNumbersDivisibleBy3
{
  public static void main(String[] args)
  {
    int prod = 1;
    for (int i = 15; i <= 35; i++)
    {
      prod = prod * i;
      if (prod % 3 == 0)
      {
        System.out.println(prod);
      }
    }
  }
}
```

Output :-
15 (Wrong)
240
4080
73440
1395360
27907200
586051200
8224512
189163776
3123732480
-1393557504

2. class SumofNumbersDivisibleBy5

```
{
  public static void main(String[] args)
  {
    int sum = 0;
    for(int i = 7; i <= 28; i++)
    {
      if(i%5 == 0)
      {
        sum = sum+i;
      }
    }
    S.O.pln(sum);
  }
}
```

output : 70

3. class productofNumbersDivisibleBy3

```
{
  public static void main(String[] args)
  {
    long product = 1;
    for(int i = 15; i <= 35; i++)
    {
      if(i % 3 == 0)
      {
        product = product * i;
      }
    }
    S.O.pln(product);
  }
}
```

→ int product :
o/p : -657548896

Long product
o/p : 363-7418400

1) Write a logic to count all the even numbers present between 10 to 20.

```
Class CountAllTheEvenNumbers
{
  public static void main (String [] args)
  {
    int Count = 0;
    for (int i = 10; i <= 20; i++)
    {
      if (i % 2 == 0)
      {
        Count++;
      }
    }
    S.O.pln (Count);
  }
}
```

Count

0

output: 6

2) Write a logic to count all the numbers which are divisible by 3 as well as to count all the no numbers which are not divisible by 3. Range 15 to 30.

```
Class CountNumbersDivisibleBy3
{
  public static void main (String [] args)
  {
    int Count1 = 0;  int Count2 = 0;
    for (int i = 15; i <= 30; i++)
    {
      if (i % 3 == 0)
      {
        Count1++;
      }
      else
      {
        Count2++;
      }
    }
  }
}
```

O/P

S.O.pln (Count1); 6

S.O.pln (Count2); 10

3. Write a logic to print alphabets A - Z

```
class printAlphabets
{
  public static void main (String[] args)
  {
    char ch = 'A';
    for (int i = 1; i <= 26; i++)
    {
      S.O.pln (ch);
      ch++;
    }
  }
}
```

ch | A | B | C |

A ++ 65 ++ 66 → B

B ++ 66++ 67 → C

O/P:
A
B
C
D
E

(or)

```
for (char ch = 'A';
     ch <= 'Z'; ch++)
{
S.O.pln (ch);
}
```

Note 1 :- System.out.println( );

This statement has two functionalities

(i) printing (ii) go to next line

Note 2 :- System.out.print( );

It has only one functionality i.e., To print and the cursor remains in the same line.

1) Write a program to print multiplication table for 7.

O/p should be below manner.

7x1 = 7
7x2 = 14
7x3 = 21
⋮
7x10 = 70

2) Write a program to print below outputs.

(a)
Aa
Ab
Ac
Ad
⋮
Az

(b)
Aa
Bb
Cc
Dd
⋮
Zz

(c)
A1
A2
A3
⋮
A10

(d)
A1
B2
C3
⋮
Z26

3) Write a program to print all even and odd numbers.

4) Write a program to count all even and odd numbers

5) what is below output.

(i)
```
int a = 5;
if (true)
{
s.o.pln(a);
}
else
{
so.pln(a++);
}
s.o.pln(a);
```

(ii)
```
int b=10;
if (b<5)
{
s.o.pln(++b);
}
else
{s.o.pln(b++)
}
s.o.pln(b)
```

(iii)
```
if (true)
{
int a =10;
s.o.pln(a);
}
else
{
s.o.pln(a);
}
```

(iv)
```
for (int i = 1; i<=10; i++)
{
s.o.pln(i);
}
s.o.pln(i);
```

(v):
```
int i;
for (int i = 1; i<=10; i++)
{
s.o.ph(i);
}
s.o.ph(i);
```

Answers for test2

```
for(i=1; i<=10; i++)
{
  int res = 7*i;
  S.O.pln("7* "+i+"="+res);
}
```

(ii) 7*i.

2(a)
```
Aa
Ab
!
Az.
```

```
char ch1 = 'A';
for(char ch2 = 'a'; ch2 <= 'z'; ch2++)
{
  S.O.print(ch1);
  S.O.pln(ch2);
}
```

(8) S.O.pln("A"+ch);

(ii)

```
Char ch1 = 'A';
    ch2 = 'a';
for(int i=1; i<=26;
    i++)
{ S.O.pln(ch1);
  S.O.pln(ch2);
}  ch2++
```

2(b)
```
Aa
Bb
Cc
!
;
Zz
```

```
Char ch1 = 'A';
Char ch2 = 'a';
for(int i=1; i<=26; i++)
{ S.O.pln(ch1);
  S.O.pln(ch2);
  ch1++;
  ch2++;
}
```

```
Char ch1 = 'A';
for(Char ch2 = 'a'; ch<='z';
    ch2++)
{ S.O.pln(ch1);
  S.O.pln(ch2);
  ch1++
}
```

2(c)
```
A1
A2
!
A10
```

```
char ch = 'A';
for(int i=1; i<=10; i++)
{
  S.O.pln(ch);
  S.O.pln(i);
}
```

2(d) A1
B2
:
Z26

```
int n=1;
Char ch = 'A';
for(int i=1; i<=26
 ; i++ )
{
 s.o.pln (ch);
 s.o.pln (n);
 ch++;
 n++;
}
```

```
Char ch = 'A';
for (int i = 1; i<=26; i++ )
{ Sop
 s.o.pln (ch);
   s.o.pln (i);
     ch++ ;
}
```

```
int n= 1;
for(ch= 'A'; ch<='Z';
 ch++)
{
 s.o.p(ch);
 s.o.pln(n);
 n++;
}
```

(3)

```
for( int i = 1 ; i <= 10) ; i+)
 { if (i%2 = =0)
   {
    s.o.pln ( i+ "is even number");
   }
   else{
    s.o.pln( i + "is odd number");
 } }
```

(4)

```
int Count1 = 0 ;
    count2 =0 ;
for( int i = 1; i <=10; i++)
 {
   if( i%2 = =0)
  {
   count1 ++;
  }
  else
   {
    Count2 ++ ;
  }
}
 s.o.pln (Count1);
 s.o.pln (Count2);
```

(5)

5(a) 5 & 5 —20
(b) 10 & 11 —(4)
(c) Error.
(d) Error (
(e) 1 2 3 4 5 6 7 8 9
10 11

(6) —(1)
(7) —(2)

Scanned by CamScanner

18/02/19 :-

Nested for loop :-

Ex:

```
SO.pln ("Start Exercise");
for (int i = 1; i <= 3; i++)
{
  for (int j = 1; j <= 3; j++)
  {
    SO.pln ("push up");
  }
  SO.pln ("Take 1 min rest");
}
SO.pln ("stop Exercise");
```

Output:

Start Exercise
push up
push up
push up
Take 1 min rest
push up
push up
push up
Take 1 min rest
push up
push up
push up
Take 1 min rest
Stop exercise

i → 1
j → 1, 2, 3, ✗

i → 2
j → 1, 2, 3, ✗

i → 3
j → 1, 2, 3, ✗

i → ✗

int i = 1

int j = 1 2 3 4 ✗

1 <= 3 ✓ push j++
2 <= 3 ✓ push j++
3 <= 3 ✓ push j++
4 <= 3 ✗ Take 1 min rest

i++ → i = 2

j = ✗ ✗ ✗ 4

1 <= 3 ✓ push j++
2 <= 3 ✓ push j++
3 <= 3 ✓ push j++
4 <= 3 ✗ (Take rest)

i++ i = 3

j = ✗ ✗ ✗ 4

1 <= 3 ✓ push j++
2 <= 3 ✓ push j++
3 <= 3 ✓ push j++
4 <= 3 ✗ Take 1 min rest

i++

4 <= 3 ✗ stop

Pattern programs :-

→ Vertical represents outer loop.

→ Horizontal represents Inner loop.

Pattern 1 :-

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

```
for (int i=1; i<=4; i++)
{
    for (int j=1; j<=4; j++)
    {
        S.O.p("* ");
    }
    S.O.pln();
}
```

pattern 2 :-

3×4 → columns, ↑ rows

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

Pattern3 : 7 x 4

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

Pattern -2 :-

```
Class pattern2
{
 public static void main
          (String[] args)
 {
 for (int i=1; i<=3; i++)
 {
  for (int j=1; j<=4; j++)
  {
   S.O.print("* ");
  }
  S.O.pln();
 }
}
```

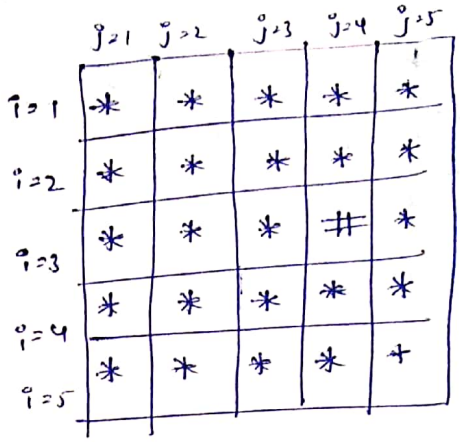
pattern 3 :-

```
class pattern3
{
 public static void main (String[] args)
 {
  for (int i=1; i<=7; i++)
  {
   for (int j=1; j<=4; j++)
   {
    S.O.print("* ");
   }
   S.O.pln();
  }
 }
}
```

19/02/19

Pattern -4 :-

```
* * * * *
* * * * *
* * * # *
* * * * *
* * * * *
```




```
class pattern4
{
  public static void main (String[] args)
  {
    for (int i=1; i<=5; i++)
    {
      for (int j=1; j<=5; j++)
      {
        if ((i==3)&&(j==4))
        {
          S.O.print('#');
        }
        else
        {
          S.O.print('*');
        }
      }
      S.O.println();
    }
  }
}
```

Pattern 5 :-

```
* * * * *
* @ * * *
* * * * *
* * * * *
* * * * *
```

```
for (int i=1; i<=5; i++)
{
  for (int j=1; j<=5; j++)
  {
    if ((i==2)&&(j==2))
    {
      S.O.p('@');
    }
    else
    {
      S.O.p('*');
    }
  }
  S.O.pln();
}
```

Pattern 6 :—

```
* * * * @
* * * * *
* * * * *
* # * * *
* * * * *
```

```
for (int i=1 ; i<=5; i++)
{
  for(int j=1;j<=5; j++)
  {
    if((i==1)&&(j==5))
    {
      s.o.p('@');
    }
    else if ((i==4)&&(j==2))
    {
      s.o.p('#');
    }
    else
    {
      s.o.p('*');
    }
  }
  s.o.pln();
}
```

Pattern 7 :—

```
* * @ * *
* * @ * *
* * @ * *
* * @ * *
* * @ * *
```

```
for (int i=1; i<=5; i++)
{
  for (int j=1; j<=5; j++)
  {
    if (j==3)
    {
      s.o.p('@');
    }
    else
    {
      s.o.p('*');
    }
  }
  s.o.pln();
}
```

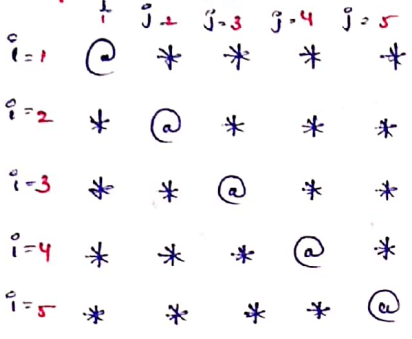
Pattern 8 :—

```
* * * * *
* * * * *
* * * * *
? ? ? ? ?
* * * * *
```

```
for (int i=1; i<=5; i++)
{
  for(int j=1;j<=5;j++)
  {
    if (i==4)
    {
      s.o.p('?');
    }
```

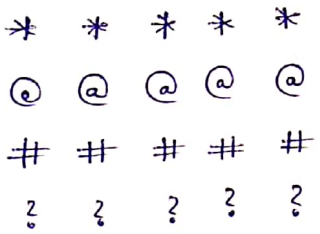
```
        else
        {
          s.o.p('*');
        }
      }
      s.o.pln();
    }
```

Pattern 9 :-




```
for (int i = 1; i <= 5; i++)
{
  for (int j = 1; j <= 5; j++)
  {
    if (i == j)
    {
      s.o.p('@');
    }
    else
    {
      s.o.p('*');
    }
  }
  s.o.pln();
}
```

Pattern 10 :-




```
for (int i = 1; i <= 5; i++)
{
  for (int j = 1; j <= 5; j++)
  {
    if (i == 1)
    {
      s.o.p('*');
    }
    else if (i == 2)
    {
      s.o.p('@');
    }
    else if (i == 3)
    {
      s.o.p('#');
    }
    else
    {
      s.o.p('?');
    }
  }
  s.o.pln();
}
```

by using switch

```
switch
{
  case 1 :
```

```
for (int i = 1; i <= 4; i++)
{
  for (int j = 1; j <= 5; j++)
  {
    switch (i):
    {
      case 1 : s.o.p ('*');
               break;
      case 2 : s.o.p ('@');
               break;
      case 3 : s.o.p ('#');
               break;
      case 4 : s.o.p ('?');
               break:
    }
  }
  s.o.pln();
}
```

dont case value

Pattern 11 :-

```
      j=1  2  3  4  5
i=1   *  *  *  *  *
i=2   *           *
i=3   *           *
i=4   *           *
i=5   *  *  *  *  *
```

Wrong

```
for (int i = 1; i <= 5; i++)
{
  for (int j = 1; j <= 5; j++)
  {
    if ((i == 1) && (j == 1))
    {
      s.o.p('*');
    }
    else if ((i == 5) && (j == 5))
    {
      s.o.p('*');
    }
    else
    {
      s.o.p(' ');
    }
  }
  s.o.pln();
}
```

Pattern 11 :-

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

```
for (int i = 1; i <= 5; i++)
{
  for (int j = 1; j <= 5; j++)
  {
    if ((i == 1) || (j == 5) || (i == 5) || (j == 1))
    {
      s.o.p('*');
    }
    else
    {
      s.o.p('□');
    }
  }
  s.o.pln();
}
```

Date: 20/02/19

Pattern 12 :-

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

	1	2	3	4
1	*			
2	*	*		
3	*	*	*	
4	*	*	*	*

i	j	Condition i >= j
1	1	1 >= 1
2	1	2 > 1
2	2	2 == 2
3	1	3 > 1
3	2	3 > 2
3	3	3 == 3
4	1	4 > 1
4	2	4 > 2
4	3	4 > 3
4	4	4 == 4

```
for (int i = 1; i <= 4; i++)
{
  for (int j = 1; j <= 4; j++)
  {
    if (i >= j)
    {
      s.o.p('*');
    }
    else
    {
      s.o.p('□');
    }
  }
  s.o.pln();
}
```

→ unnecessary (or) optional

Pattern 13 :-

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

	1	2	3	4
1	*	*	*	*
2		*	*	*
3			*	*
4			*	*

i	j
1	1
1	< 2
1	< 3
1	< 4

2 == 2, 2 < 3, 2 < 4 | 3 == 3, 3 < 4 | 4 == 4

i <= j

```
for (int i = 1; i <= 4; i++)
{
  for (int j = 1; j <= 4; j++)
  {
    if (i <= j)
    {
      S.O.p('*');
    }
    else
    {
      S.O.p('□');
    }
  }
  S.O.pln();
}
```

Pattern 14 :-

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

i	j
1	< 4
2	< 3
2	< 4

3 < 2, 3 == 3, 3 < 4

4 1 4 3
4 2 4

1

i	j
1	< 4
2	< 3
2	< 4
3	< 2
3	== 3
3	< 4
4	> 1
4	> 2
4	> 3
4	== 4

Not applicable (<, >, =)

```
for (int i = 1; i <= 4; i++)
{
  for (int j = 1; j <= 4; j++)
  {
    if (
```

	j=1	2	3	4
i = 4				*
i 3			*	*
i 2		*	*	*
i 1	*	*	*	*

i	j
4	== 4
3	== 3
3	< 4
2	== 2
2	< 3
2	< 4

1 = 1, 1 < 2, 1 < 3, 1 < 4

i <= j

i	j
4	4 1
4	4 4
4	3
4	4

```
for (int i = 4; i >= 1; i--)
{
  for (int j = 1; j <= 4; j++)
  {
    if (i <= j)
    {
      s.o.p('*');
    }
    else
    {
      s.o.p('□');
    }
  }
  s.o.println();
}
```

Steps :-

Pattern 15 :-

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

```
for (int i = 1; i <= 4; i++)
{
  for (int j = 4; j >= 1; j--)
  {
    if (i <= j)
    {
      s.o.p('*');
    }
    else
    {
      s.o.p('□');
    }
  }
  s.o.println();
}
```

```
    4  3  2  1
1   *  *  *  *
2   *  *  *
3   *  *
4   *
```

i	j
1	1
2	1
2	2
2	3

i	j
1 < 4	
1 < 3	
1 < 2	
1 == 1	
2 < 4	
2 < 3	
2 == 2	
3 < 4	
3 <= 3	
4 == 4	

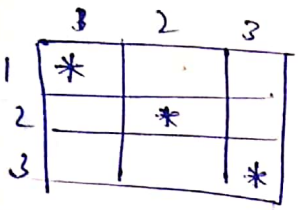
another approach :-

```
    1  2  3  4
4   *  *  *  *
3   *  *  *
2   *  *
1   *
```

i >= j

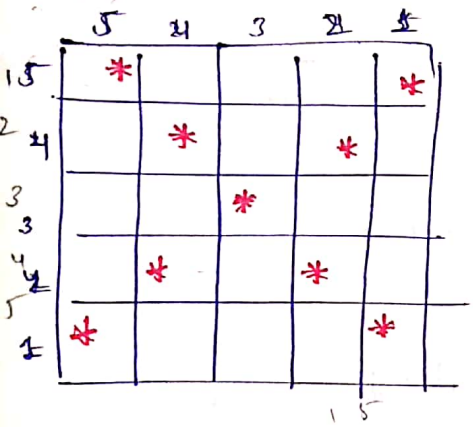
Pattern 16 :-

```
*
 *
  *
```

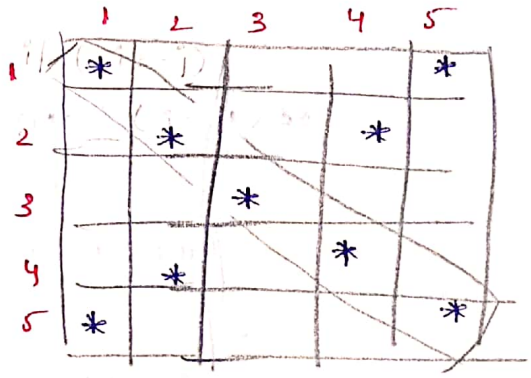


```
for (int i = 1; i <= 3; i++)
{
  for (int j = 1; j <= 3; j++)
  {
    if (i == j)
    {
      s.o.p('*');
    }
    else {
      s.o.p('□');
    }
  }
  s.o.println();
}
```

Pattern 17 :-



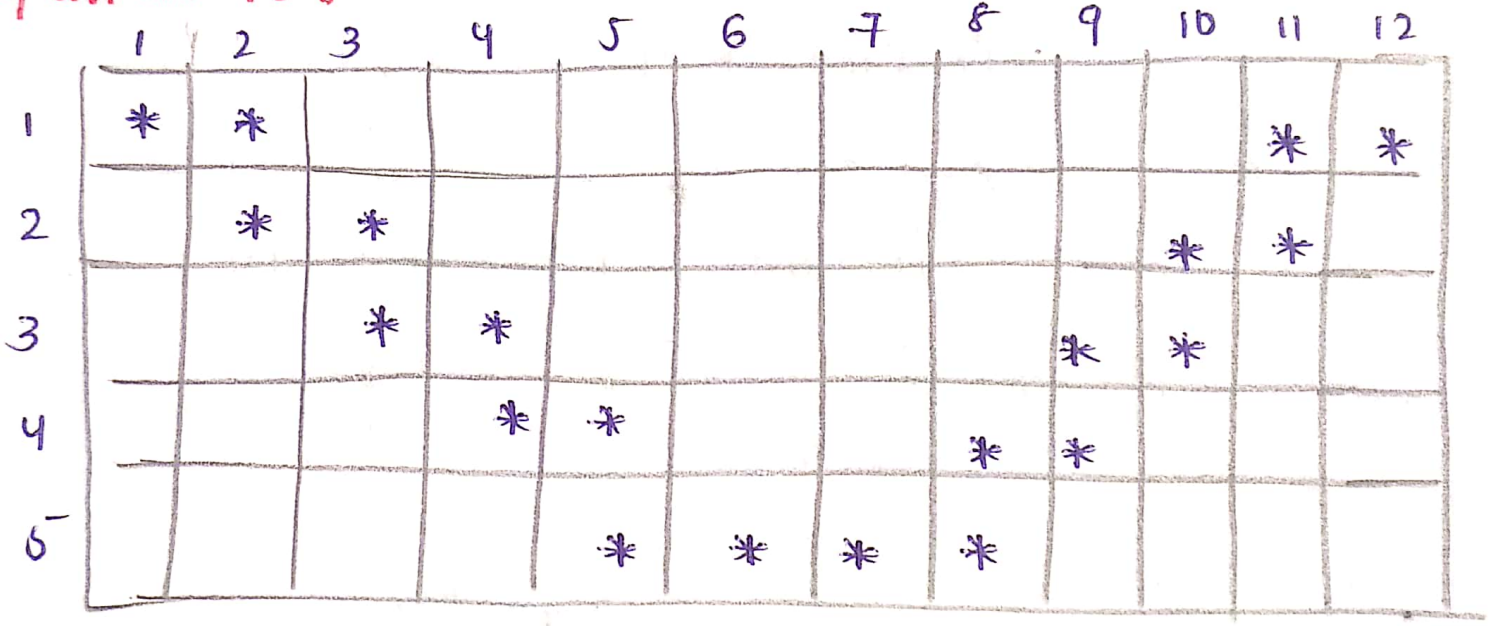
i	j
1	5
2	4
3	3
4	2
5	1



i	j
1	5
2	4
4	2
5	1

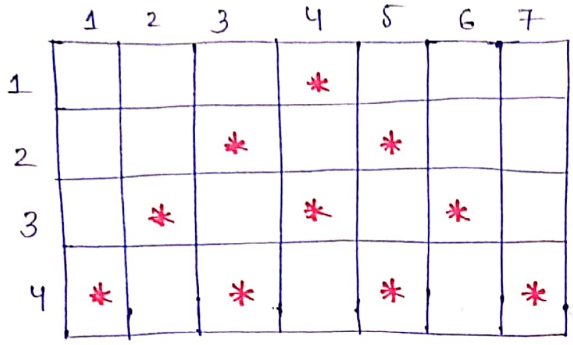
```
for (int i = 1; i <= 5; i++)
{
  for (int j = 1; j <= 5; j++)
  {
    if (i == j || i+j == 6)
    {
      s.o.p('*');
    }
    else
    {
      s.o.p('□');
    }
  }
  s.o.println();
}
```

Pattern 18 :-



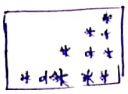
21/02/19:

Pattern 19 :-



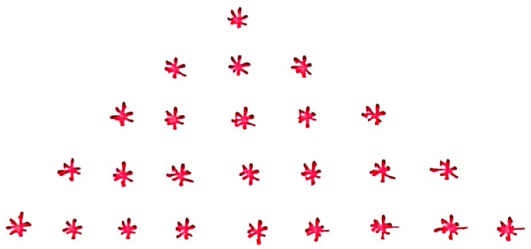

```
for(int i=4; i>=1; i--)
{
  for(int j=1; j<=4; j++)
  {
    if(i<=j)
    {
      s.o.p("*□");
    }
    else
    {
      s.o.p('□');
    }
  }
  s.o.pln();
}
```

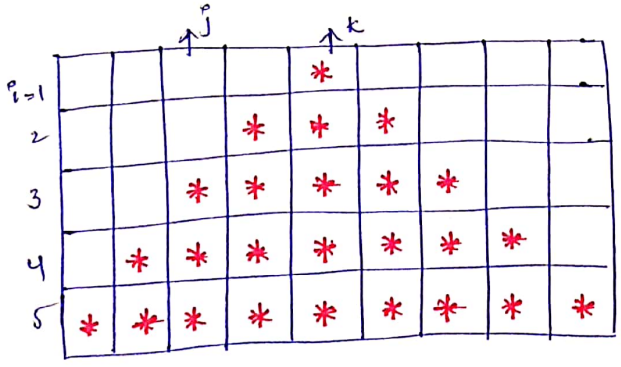
```
if((i+j==5)||(i+j==7)||(i+j==9)||
        (i+j==11))
{
  s.o.p('*');
}else{
    s.o.p('□');
}
```



→ from this program we write the logic but instead of printing single space we print space along with star then we get required pattern.

Pattern 20 :-





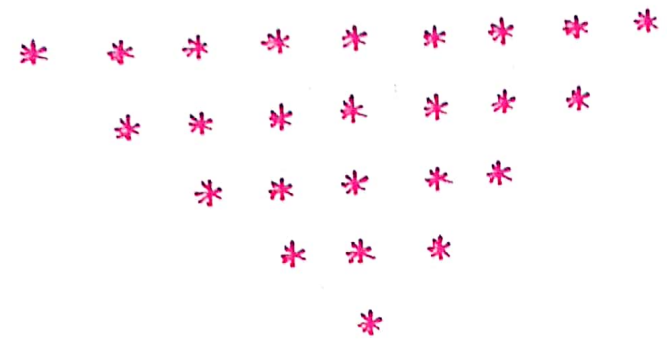

```
int space=4;
int star=1;
for(int i=1; i<=5; i++)
{
  for(int j=1; j<=space; j++)
  {
    s.o.p(' ');
  }
  for(int k=1; k<=star; k++)
  {
    s.o.p('*');
  }
  s.o.pln();
  space--;
  star+=2;
}
```

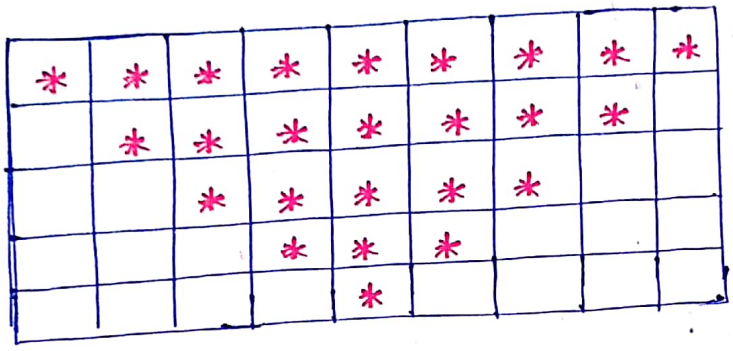
→ Outerloops are considered as two one is for spaces, another is for stars

→ Spaces are decreased by one (1)

→ stars are increased by 2 (two)

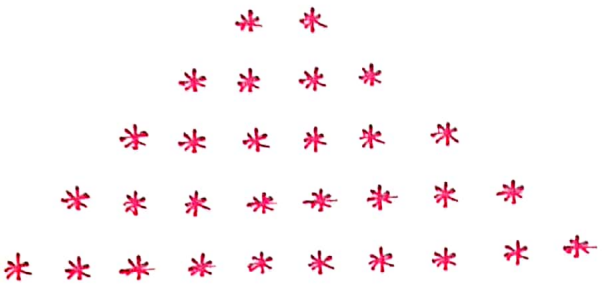
Pattern Q1 :-

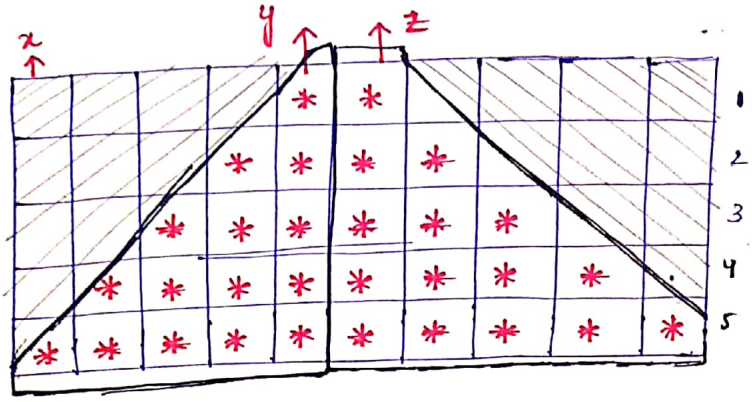




```
int space = 0;
int star = 9;
for(int i = 1; i <= 5; i++)
{
  for(int j = 1; j <= space; j++)
  {
    S.o.p(" ");
  }
  for(int k = 1; k <= star; k++)
  {
    S.o.p("*□");
  }
  S.o.pln();
  space++;
  star -= 2;
}
```

Pattern 22 :-



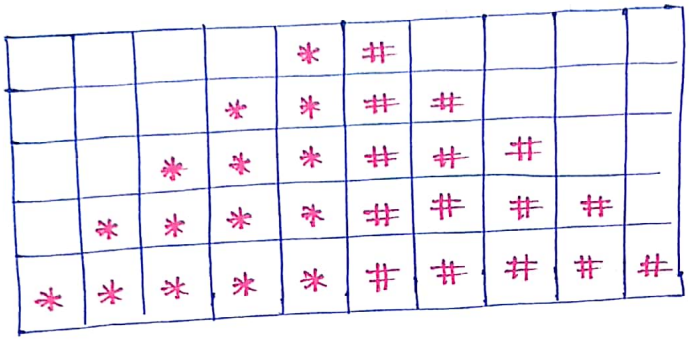


We use 4 for loops.
one for outer loop.
one for space
one for stars.
another for stars.

```
int space = 4;
int star = 2;
for (int i = 1; i <= 5; i++)
{
  for (int j = 1; j <= space; j++)
  {
    for (int x = 1; x <= 5; x++)
    {
      S.O.p(' ');
    }
  for (int k = 1; k <= star; k++)
  {
    S.O.p('*');
  }
    S.O.pln();
    space --;
    star += 2;
  }
}
```

Pattern 23 :-

```
        * #
      * * # #
    * * * # # #
  * * * * # # # #
* * * * * # # # # #
```



```
int space = 4;
int star = 1;
int hash = 1;
for (int i = 1; i <= 5; i++)
{
  for (int j = 1; j <= space; j++)
  {
    S.O.p(' ');
  }
  for (int k = 1; k <= star; k++)
  {
    S.O.p('*');
  }
  for (int l = 1; l <= hash; l++)
  {
    S.O.p('#');
  }
  S.O.pln();
  space--;
  star++;
  hash++;
}
```

21/02/19

Write a program to check whether the given number is prime number or not.

```
Class primenumber
{
  public static void main (String[] args)
  {
     int n= 5;
     int numofFactors = 0;
     for (int i = 1; i <= n; i++)
     {
        if (n%i == 0)
        {
           numofFactors++;
        }
     }
     if (numofFactors == 2)
     {
       S.o.pln (n+" is primenumber");
     }
     else
     {
       S.o.pln (n+" is not primenumber");
     }
  }
}
```

n = 5

i = 1

1 <= 5 (T)

5%1 == 0 (T)

numofFactors++;

0 1 2

2 <= 5 (T)

5%2 != 0 (F)

3 <= 5 (T)

5%3 != 0 (F)

4 <= 5 (T)

5%4 != 0 (F)

5 <= 5 (T)

5%5 == 0 (T)

numofFactors++ (2

i++ (6 <= 5) (F)

Above logic works only for one case. (n >= 2) if n value starts from 2 & above.

If n value is 1 our logic gives wrong result. If n value is 0 & negative number. It is a invalid number.

→ Above logic should be modify to support all the three cases.

```
int n = 1;
int numofFactors = 0;
for(int i = 1; i<=n; i++)
{
    if(n%i ==0)
    {
        numofFactors++;
    }
}
if(numofFactors ==2)
{
    S.O.pln(n+" is prime number");
}
else
{
    S.O.pln(n+" is not prime number");
}
else if(n==1)
{
    S.O.pln("1 is neither a primenumber nor a prime
                number");
}
else
{
    S.O.pln("Invalid number");
}
}
```

→ if(n>=2) {

Write a program to reverse a number.

/ → quotient

% → remainder

```
Class ReverseNumber
{
public static void main (String[] args)
 {
    int n = 456;
    int ln = 0;
    int rev = 0;
    for (int i = 1; i <= 3; i++)
    {
        ln = n % 10;
        rev = rev * 10 + ln;
        n = n/10;
    }
    S.O.pln (rev);
 }
}
```

[ln last number]

n → 456

ln → 0

rev → 0

n = 456 ln = 0 rev = 0

→ n%10 = 456/10 = 6 (ln)

→ rev * 10 + ln

0 * 10 + 6 = 0+6 = 6 (rev)

n/10 => 456/10 = 45 (n)

→ n%10 = 45/10 = 5 (ln)

rev * 10 + ln

6 * 10 + 5 = 65 (rev)

n/10 = 45/10 = 4 (n)

n%10 = 4/10 = 4 (ln)

rev * 10 + ln

65 * 10 + 4 => 650 + 4 = 654 (rev)

n/10 = 0.4 → decimal

integer

n = 0

↓

when divide two integer the result will retain only integer part decimal part will be removed and integer is the answer.

for only quotient → numerator < denominator

4 < 10 → numerator

4 ← will be equal to remainder

num < deno

1st loop → ln → 6 (remainder) 10)456(45 / 450 / 6

rev => rev * 10 + ln

0 x 10 + 6 → 6

n → 456/10 = 45 (n)

2nd loop → ln → 5 10)45(4 / 40 / 5

rev → rev * 10 + ln

(65) 6 * 10 + 5 = 65

n → 45/10 = 4 (n)

3rd loop → ln → 4/10 = 4.

rev = rev * 10 + ln = 65 x 10 + 4

= 654

654

n → 4/10 = 0.4

.0

n = 0

1) Write a logic to check whether given number is palindrome or not (545 & 769)

2) Write a program to check whether given number is armstrong number or not

```
class palindrome
{
psvm(String[] args)
{
 int n = 545;
 int temp = n;
 int ln = 0;
 int rev = 0;
 for(int i = 1; i<=3; i++)
 {
   ln = n%10;
   rev = rev * 10 + ln;
    n = n/10
 }
 if (rev == temp)
 {
  s.o.p("given num is palindrome");
 }
 else
 {
  s.o.p("not a palindrome
 }
}
}
```

```
class Armstrong
{
 psvM(String[] args)
 {
   int n = 153;
   int temp = n;
   int ln = 0;
   int sum = 0;
   for(int i = 1; i<=3; i++)
   {
     ln = n%10;
     sum = (ln * ln * ln) + sum;
      n = n/10;
   }
   if(sum == temp)
   {
    s.o.pln(" Armstrong");
   }
   else
   {
    s.o.pln(" not Armstrong");
   }
 }
}
```

27/02/19

Reading Input during Runtime :-

1) Inorder to read input during runtime we must make use of inbuilt Scanner class

2) Scanner class is present within java.util package. (java-folder)

3) By using operations (Methods) of Scanner class, we can read
   (i) String data → next();
   (ii) Integer data → nextInt();
   (iii) Decimal data → nextDouble();

4) To use this Scanner class in our program we must imported
   ** import java.util.Scanner;

5) To use operations of Scanner class, we must create an object.
   ** Scanner scan = new Scanner(System.in);

Write a program to read String data, Integer data and decimal data. and print it.

```
import java.util.Scanner;
class ReadData
{
public static void main (String[] args)
 {
   Scanner scan = new Scanner(System.in);
   System.out.println("Enter String data");
    String a = scan.next();
  System.out.println("enter Int data");
  int b = scan.nextInt();
```

```
        System.out.println("Enter decimal data");
        double c = Scan.nextDouble();
        System.out.println("Enter data is are....");
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}
```

Output:-

```
Enter String data
Sanju
Enter Int data
23
Enter decimal data
5.1
```

```
Entered data's are....
Sanju
23
5.1
```

Write a program to read your name, age, height during runtime and print those information.

Biodata.java

```
import java.util.Scanner;
class Biodata
{
 psvm(String[] args)
 {
  Scanner scan = new Scanner(System.in);
  S.o.pln("Enter the name");
  String a = Scan.next();          → a can be written as name also
  S.o.pln("Enter the age");
  int b = Scan.nextInt();          → b can be written as age
  S.o.pln("Enter the height");
  double c = Scan.nextDouble();    → c can be written as height
```

```
        System.out.println ("Entered Data's are...");
        System.out.println (a);
        System.out.println (b);
        System.out.println (c);
    }
}
```

Output :-

```
Enter the name
Sandhya
Enter the age
21
Enter the height
5.1
```

```
Entered Data's are :...
Sandhya
21
5.1
```

Note :-

→ within Scanner class there is no operation (method) to read character data.

→ Inorder to read a character follow below approach.

Step 1 : Read character in the form of string.

Step 2 :- Go to the string get the character by using

charAt(index)

```
S.O.pln ("Enter character");
String s1 = scan.next();        ← S
```

s1 | "S"

```
char ch = s1.charAt(0);
```

ch | 'S'

```
S.O.pln (ch);
```

Write a program to find sum of integer value and decimal value read the value during runtime. print the result

```
import java.util.Scanner;
class Addition
{
  public static void main(String[] args)
  {
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter Integer value");
    int a = scan.nextInt();
    System.out.println("Enter decimal value");
    double b = scan.nextDouble();
    double c = a+b;
    System.out.println("Addition value"+c);
    double c= scan.nextDouble();
  }
}
```

2) Write a program to calculate simple Intrest and print the result

$$S.I = \frac{P * T * R}{100}$$

```
import java.util.Scanner;
class SimpleInterest
{
  public static void main(String[] args)
  {
    Scanner scan = new Scanner(System.in).
    System.out.println("Enter the principal");
    float p = scan.nextFloat();   (or) int p = scan.nextInt();
    System.out.println("Enter Time period");
```

```
float
int t = Scan.nextFloat();   (or) int t = Scan.nextInt();
System.out.println("Enter the Rate of Interest");
int r = scan.nextInt();   (or) in float r = Scan.nextFloat();
int si = (p*t*Rr)/100;
System.out.println("The simple Interest is : "+si);
}
}
```

output :-

```
Enter the principal
200
Enter the Timeperiod
3.5
Enter the Rate of Intrest
2.
```

The simple Interest
14.

01/03/19

Methods :-

→ Methods are used for performing operation

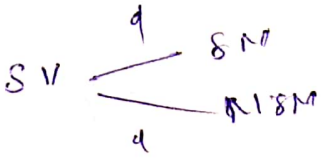
Note :- A program not only can have main method rather programm also can define a method.

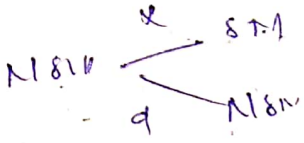
→

```
Class Sample
{
   public static void main (String[] args)
   {
      S.O.pln ("main starts");
      move();
      S.O.pln ("main ends");
   }
  public static void move()
   {
      S.O.pln ("move North");
   }
}
```

output :-

main starts
~~main ends~~
move North
main ends





```
class demo
{
public static void main (String[] args)
{
S.O.pln ("main starts");
punch();
S.O.pln ("main ends");
}
public static void punch()
{
S.O.pln ("punch starts");
kick();
S.O.pln ("punch ends");
}
public static void kick()
{
S.O.pln ("kick starts");
S.O.pln ("kick ends");
}
}
```

Output :—

```
main starts
punch starts
punch ends
kick starts
kick ends
punch ends
main ends
```

→ call the method name in the main method then the method will be executed.

(i) Every method should be called in order to execute.

(ii) Methods are called by using method names.

(iii) Methods are called from another method.

(iv) After execution of a method control returns back to calling method.

(v) Methods can be called multiple times.

02/03

Types of Variables :-

(i) There are two types of variables based on declaration.

(a) Local variable (b) Global variable

(a) Local variable :-

(i) Local variables are declared within the scope of the method and can we used only within the scope of same method.

```
Class Run
{
  public static void main (String[] args)
  {
    S.O.pln ("Main starts....");
    int a = 10; →//Local variable to main method
    S.O.pln (a);
    walk ();
    S.O.pln ("Main ends.....");
  }
  public static void walk ()
  {
    S.O.pln ("Walk starts....");
    int b = 20; →//local variable to walk method
    S.O.pln (b);
    S.O.pln ("Walk ends....");
  }
}
```

Output :-

```
Main starts....
10
Walk starts....
20
Walk ends....
Main ends....
```

Note:-

All local variables must be Initialize before its usage.

(b) Global variables :-

(i) Variables which are declared within the scope of class are called Global variables.

(ii) There are two types of Global Variables

1. Static variable
2. Non Static variable

```
class Test
{
    static int i = 10;   //its a global variable can be used in all the methods
    public static void main (String[] args)
    {
      S.O.pln ("main starts");
      S.O.pln (i);
      move();
      S.O.pln ("main ends");
    }
    public static void move()
    {
       S.O.pln ("move starts");
       S.O.pln (i);
       S.O.pln ("move ends");
    }
}
```

Output :-

```
main starts
10
move starts
10
Move ends
main ends.
```

→ Initialization of global variables is not compulsary. Incase programmer does not initialize global variables, it gets initialize to default value by compiler.

```
class Test
{
 public static byte b;
        static short s;
        static int i;
        static long l;
        static float f;
        static double d;
        static char c;
        static boolean bl;
        static String sl;
       public static void main(String[] args)
       {
         S.O.pln("main starts");
         S.O.pln(b);
         S.O.pln(s);
         S.O.pln(i);
         S.O.pln(l);
         S.O.pln(f);
         S.O.pln(d);
         S.O.pln(c);
         S.O.pln(bl);
         S.O.pln(sl);
         S.O.pln("main ends");
       }
}
```

Output:-

main starts
0
0
0
0
0.0
0.0
[ ]
false
null
main ends

Integer type of data value

float type of value

empty character

boolean true/false

String.

→ Initialization of global variables is not compulsary. Incase programmer does not initialize global variables, it gets initialize to default value by compiler.

```
class Test
{
public static byte b;
       static short s;
       static int i;
       static long l;
       static float f;
       static double d;
       static char c;
       static boolean bl;
       static String sl;
public static void main(String[] args)
{
  S.O.pln("main starts");  → main starts
  S.O.pln(b);   → 0
  S.O.pln(s);   → 0
  S.O.pln(i);   → 0
  S.O.pln(l);   → 0
  S.O.pln(f);   → 0.0
  S.O.pln(d);   → 0.0
  S.O.pln(c);   → [ ]
  S.O.pln(bl);  → false
  S.O.pln(sl);  → null
  S.O.pln("main ends"); → main ends
}
}
```

output:-

Integer type of data value

float type of value

empty character

boolean true/false

string.

Types of Methods :-

(i) Method with Argument

(ii) Method with Returntype

(i) Method with Argument :-

→ There are some operations which demands additional inputs inorder to perform operation. These operations are represented by using method with Argument.

→ Arguments part of method signature

Syntax: Access specifiers Access Modifiers void methodname (arguments)

→ Arguments are nothing but variable declaration (argument type argument name)

→ Arguments are also called as local variables, hence arguments must be initialited when method is called.

→ Arguments are used within method implementation to perform operation.

class MethodWithArgPgm

```
{
  public static void main(String[] args)
  {
    test1(8);
    test2(5.6, 9);
    test3("raja");
  }
}
```

```
public static void test1(int x)
{
  ------
  ------
}
```

x | 8

```
public static void test2(double j, int i)
{
  ------
  ------
}
```

j | 5.6 i | 9

```
public static void test3(String y)
{
  ------
  ------
}
```

y | "raja"

04/03.

(ii) Method with Returntype :-

→ Return type specifies type of data return from the method.

→ use Return (return) keyword to return the data.

→ return keyword is compulsary if return type is other than void.

Class MethodWithReturntype

{

Public static void main (String[] args)

{

int i = test1();

i 8

double j = test2();

j 7.9

char c = test3();

c 8

}

public static int test1();

{

return 8;

}

return type

return keyword

public static double test2();

{

return 7.9;

}

public static char test3();

{

return 'S';

}

}

Method Overloading :-

→ Process of developing multiple methods with the same method name but with different arguments is called "Method Overloading."

Rules for Arguments :-

(i) Between methods no. of arguments should be different. (or)

psv move (int i, int j)

(or) psv move (int i, int j, int k)

(ii) Between methods datatype of argument should be different.

psv move (int i, int j)

(or) psv move (int i, double j)

(iii) Between methods position of argument datatype should be different

psv move (int i, double j)

psv move (double i, int j)

Program :-

```
class MethodOverloading
{
  public static void main (String[] args)
  {
  public static void move (int i, int j)
  {
    S.O.pln("Inside move method with int, int args");
  }
  public static void move(int i, double j)
  {
    S.O.pln ("Inside move method with int, double args");
  }
  public static void move (double i, int j)
  {
    S.O.pln ("Inside move method with double, int args");
  }
```

```
public static void move (int i, int j, int k)
{
  S.o.pln ("Inside move method with int, int, int args");
}
public static void main (String[] args)
{
  move (5,7);
  move (5, 2.5);
  move (5.5, 6);
  move (5,6,7);
}
```

output :-

Inside move method with int, int args
Inside move method with int, double args
Inside move method with double, int args
Inside move method with int, int, int args.

* We go for method overloading when we want to perform same operation in multiple ways.
* With the help of method overloading we achieve flexibility. i.e., user can perform operation in user desired manner.

Q: real time example to explain multiple ways as well as flexibility.

Mobile charging :-

→ charging with → charger connected with board current board
→ using powerBank
→ using Booster
→ Differently used with normal wires to connect tubelight wires.
→ without Adapter connected with USB cable connected with /t laptop.

Door locks :-

→ Normal way → hanger doors → password doors
Knee locks
→ Sensor doors → using chip Doorchip.lock
(eye scanning & finger prints)

Programming example :-

Write a program to perform addition operation in four different ways.

```
Class AdditionOperation
{
  public static void add (int i, int j)
  {
    S.O.pln (" Addition of two Integer values "); / S.O.pln(i+j);
  }
  public static void add (int i, double j)
  {
    S.O.pln (" Addition of one integer value and one double value ");
                                    / S.O.pln(i+j)
  }
  public static void add ( double i, double j )
  {
    S.O.pln (" Addition of two double values "); / S.O.pln(i+j)
  }
  public static void add ( double i, int j )
  {
    S.O.pln (" Addition of one double value and one integer value");
                                    / S.O.pln(i+j);
  }
  public static void main (String[] args)
  {
     add ( 5.4, 5.11);
     add ( 22, 23);
     add ( 22, 5.4);
     add ( 5.11, 23);
  }
}
```

output :-

Addition of two double values

Addition of two Integer values

Addition of one integer value and one double value

Addition of one double value and one Integer value.

(ii)

5.61

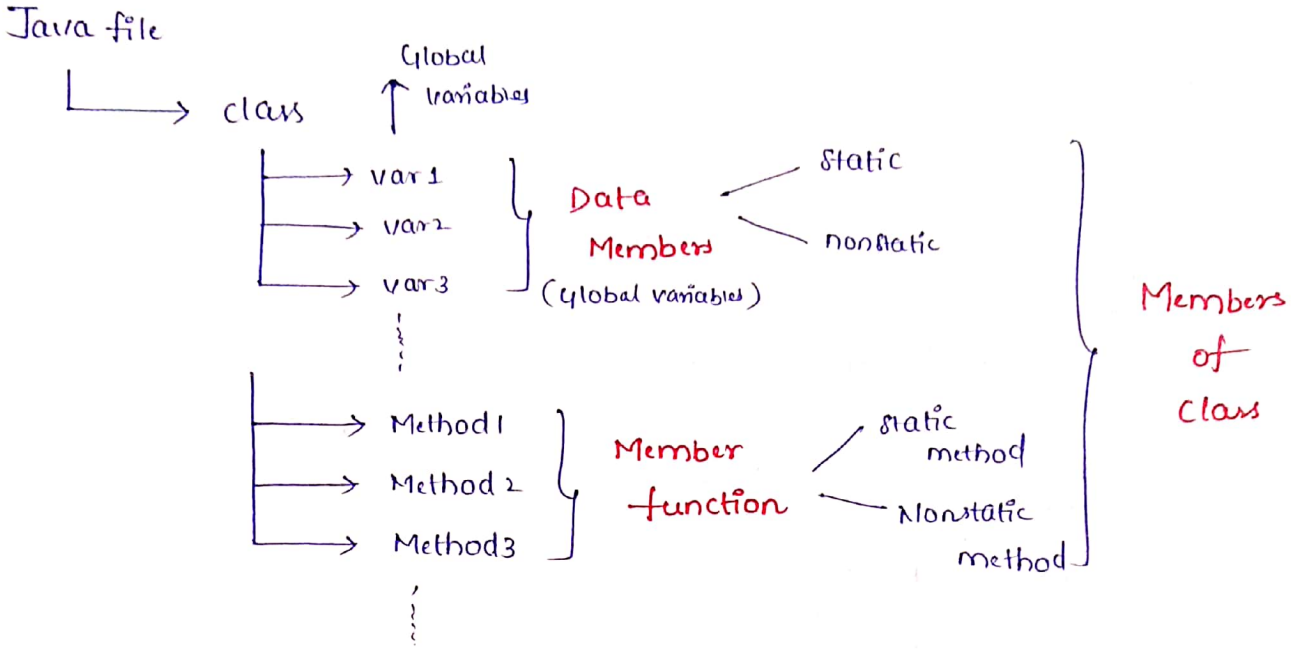
45

27.5

28.11

05/03/19

Hierarchy of Java program :-



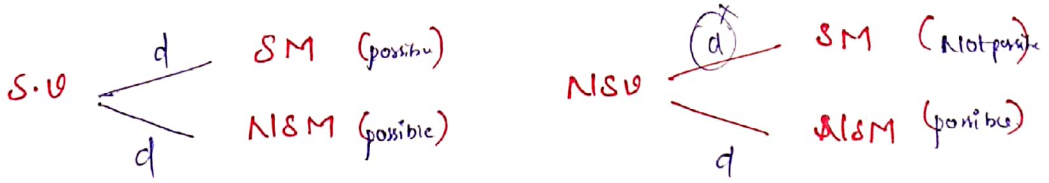
Accessing Datamembers within Memberfunction :-

(i) Static variable can be directly access within static method.

(ii) Static variable can be accessed directly within non static method

(iii) Nonstatic variable cannot be accesse directly acces from static Method

(iv) Non static variable can be directly access within Non static method.




```
class A
{
    int i = 10;
    static int j = 20;
    psvm public static void main (String[] args)
    {
        s.o.pln(i);
        s.o.pln(j);
    }
}
```

→ Error.

Reason:

Non static variable cannot directly access from static method.

(v) Non-static variable can be accessed within static method through object.

```
class Demo
{
    static int i = 60;
    int j = 70;
    public static void main (String[] args)
    {
        S.o.pln (" main starts ... ");
        S.o.pln (i);
        Demo o1 = new Demo();
        S.o.pln (o1.j);
        S.o.pln (" main ends ... ");
    }
}
```

object ←

Output :—

```
Main starts
60
70
main ends.
```

Program 2 :—

```
class Run
{
    static int x = 55;
    static double y = 3.4;
    int j = 60;
    double i = 1.2;
    public static void main (String[] args)
    {
        S.o.pln (" main starts ... ");
        S.o.pln (x);
        S.o.pln (y);
        Run o1 = new Run();
        S.o.pln (o1.j);
    // Run o2 = new Run(); // single object of 2 objects are possible).
        S.o.pln (o1.i); (or) s.o.pln(o2.i)
        s.o.pln ("main ends");
    }
}
```

program:—

```
class Test
{
  public static void move()
  {
    S.O.pln("move 5 kms ...");
  }
  public void walk()
  {
    S.O.pln("walk 10 kms ...");
  }
  public static void main(String[] args)
  {
    S.O.pln("main starts...");
    Test.move();
    Test o1 = new Test();
    o1.walk();
    S.O.pln("main ends ...");
  }
}
```

output:

```
Main starts
move 5 kms
Walk 10 kms
Main ends.
```

→ Rules that are applicable for static variable and non static variable same rules are applicable for static methods and Non static method.

06/03

Execution process :-

→ When Java program is executed two memory Areas are created

(i) Stack Area

(ii) Heap Area

(i) Stack Area :-

→ It's also called as Execution Area.

→ Every method upon call should enter Stack Area for Execution.

→ Every method should exit from stack area after execution.

→ All local variables are stored / created within Stack Area.

(ii) Heap Area :-

→ It is also called as storage Area.

→ Static members are loaded into static pool Area by class loader.

→ NonStatic Members are loaded into object during object creation.

→ All Global variables are area stored within HeapArea.

→ After execution of all the methods JVM invokes Garbage Collector and exit from StackArea.

→ Garbage Collector functionality is to clean HeapArea.

Execution diagram for Run program with detailed Explanation

ie
L Area

(x) Execution done by main method and then comes out of Stack Area.

(xi) Execution proces also completed then JVM calls the Garbage Collector and then JVM also comes out of Stack Area.

(xii) Garbage Collector is used to clean the Heap Area.

new :-

new operator can operate three operations. +1 operation

(i) create an object

(ii) load all the non static members.

(iii) Gives the address of an object → call the constructor and initialize the current object value.

& store the address of t(object name)

(i) JVM enters first into the stack Area

(ii) JVM calls the class loader.

(iii) class loader is used to load all the static members. & then comes out from stack Area.

(iv) Static members & Nonstatic members are stored in the Heap Area.

(v) All static members (static methods & static variables) are stored at static pool Area.

(vi) Execution starts from main method for static members.

(vii) All nonstatic members (Nonstatic methods & nonstatic variables) are stored at by creating an object.

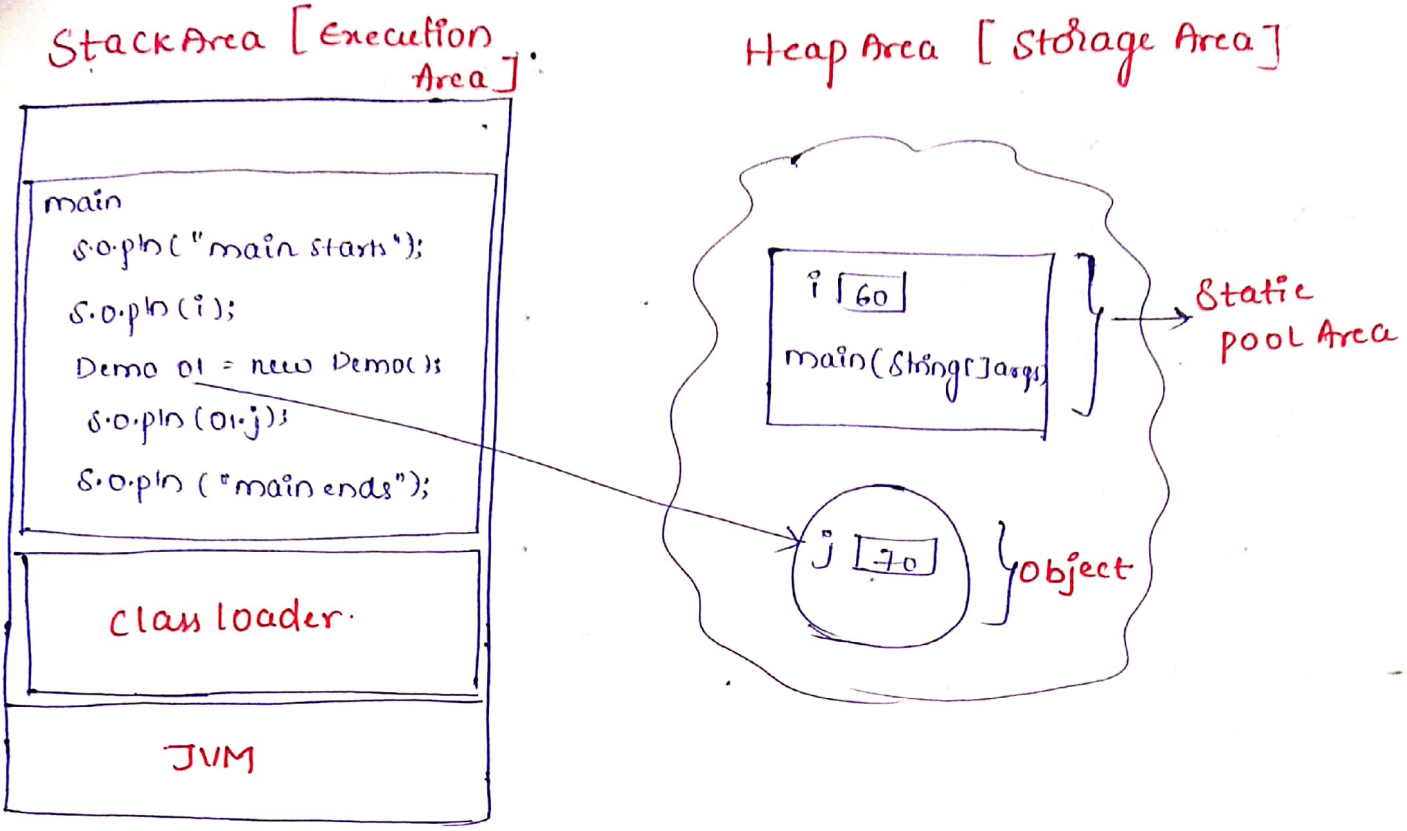
Syntax :- Classname objectname = new Classname();

then continue the execution process for nonstatic members.

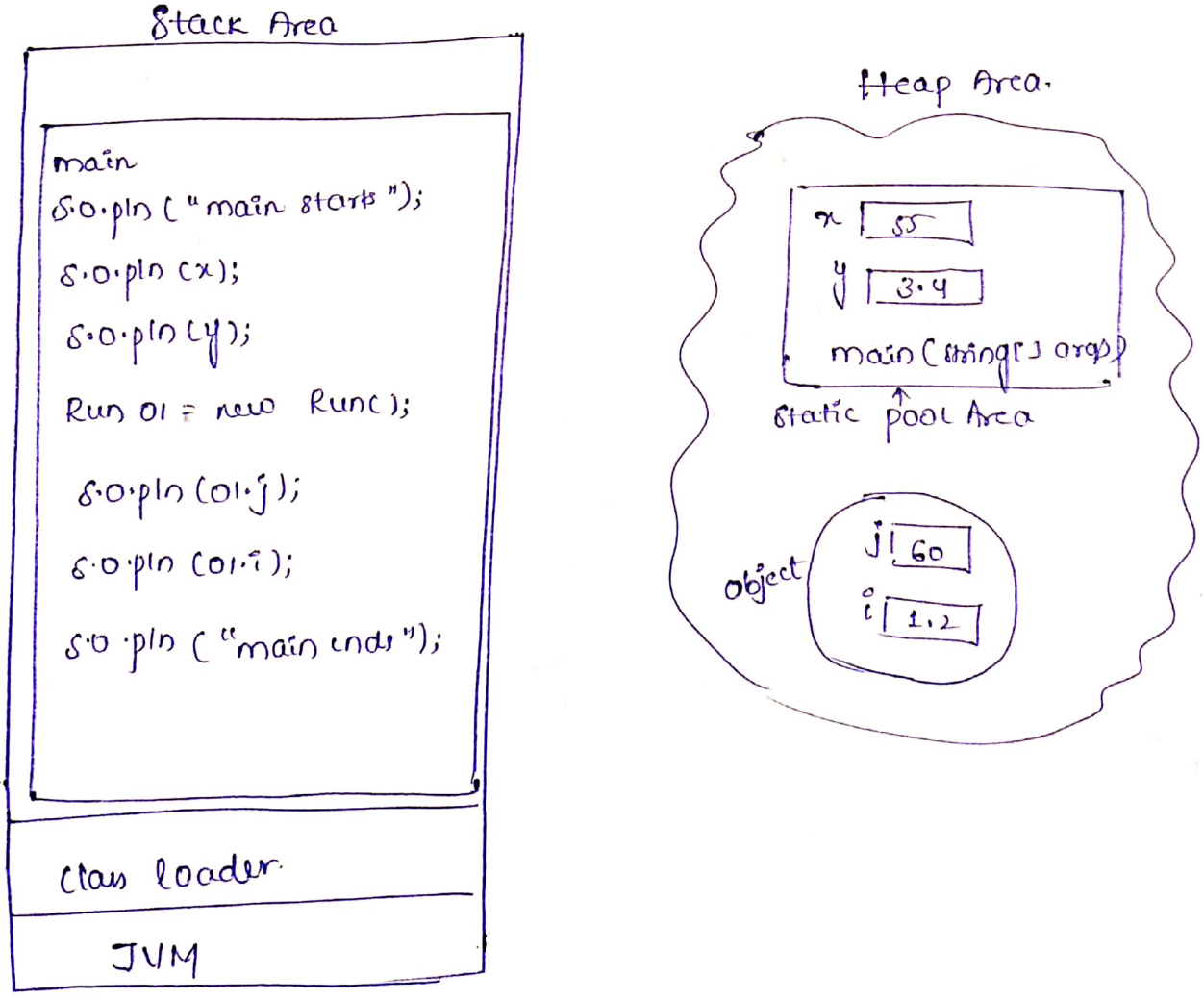
(viii) In this syntax objectname is creating an object and it gives address of an object. object name can also be called as reference variable.

(ix) Variable values first search in stackArea & checks the value is here or not and then search in Heap Area.

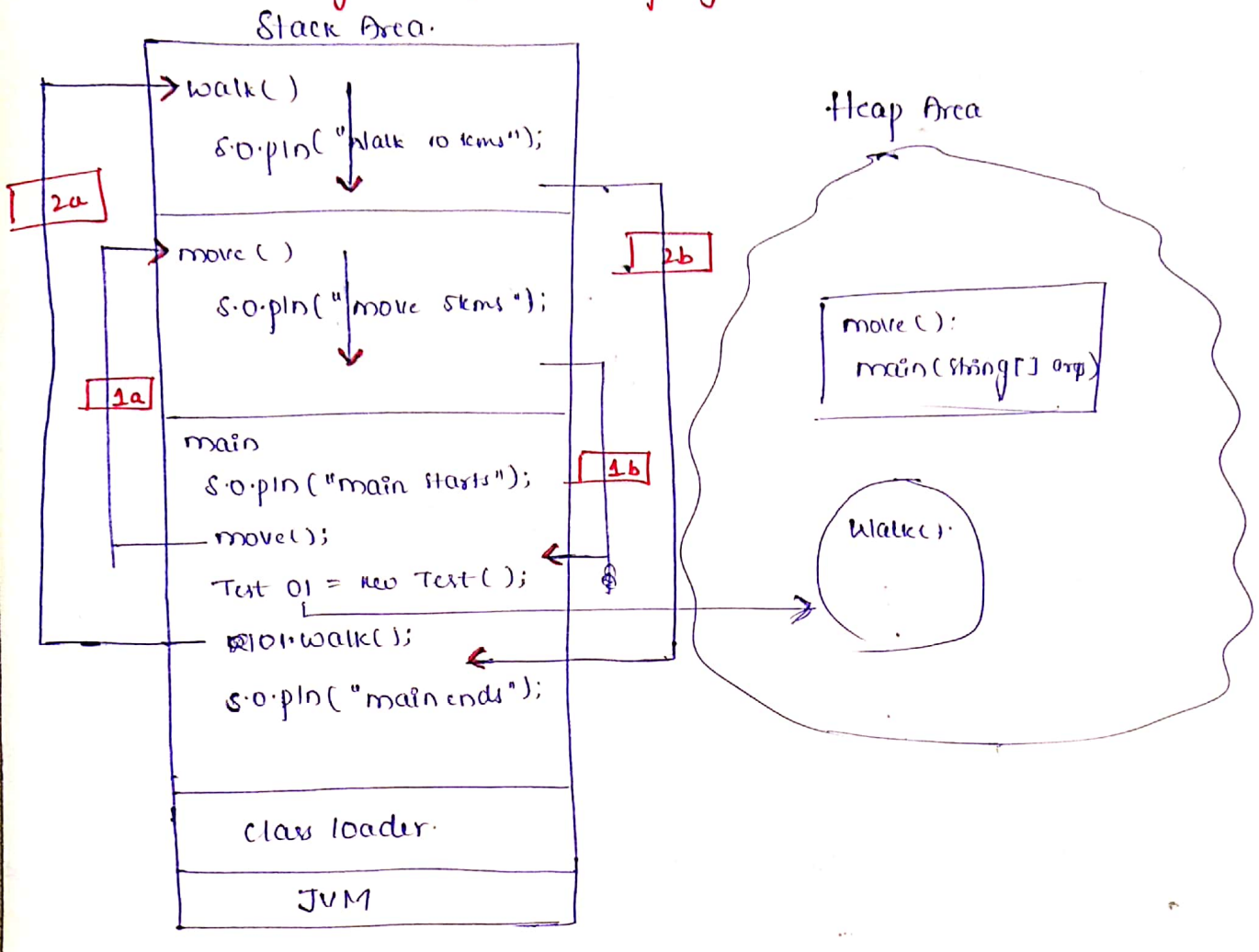
ecution diagram for Demo program :-



Execution diagram for Run program :-



Execution diagram for Test program :-



08/03

Copies of Members of class :-

→ We can have only single copy of static members because class loader gets executed only once for one class.

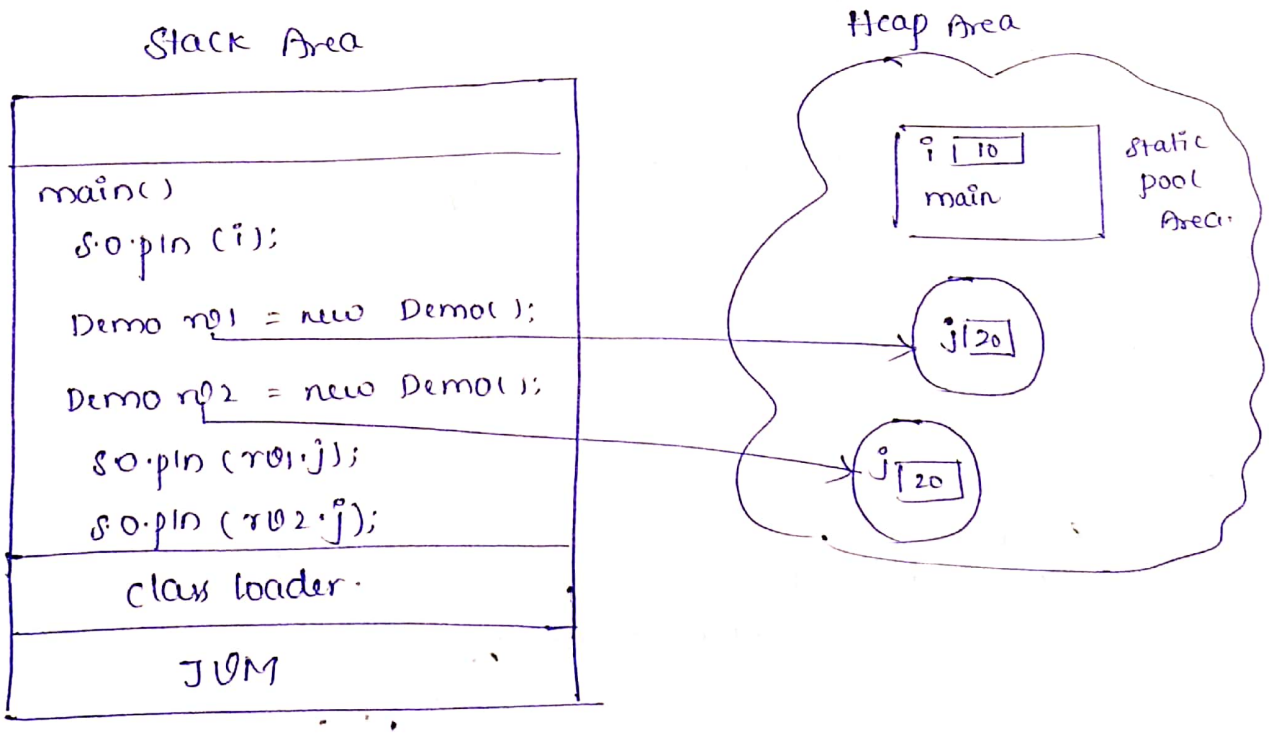
→ We can have multiple copies of nonstatic members. because multiple objects can be created by programmer.

** Note :-

Everytime non static members gets loaded whenever object is created.

Program :-

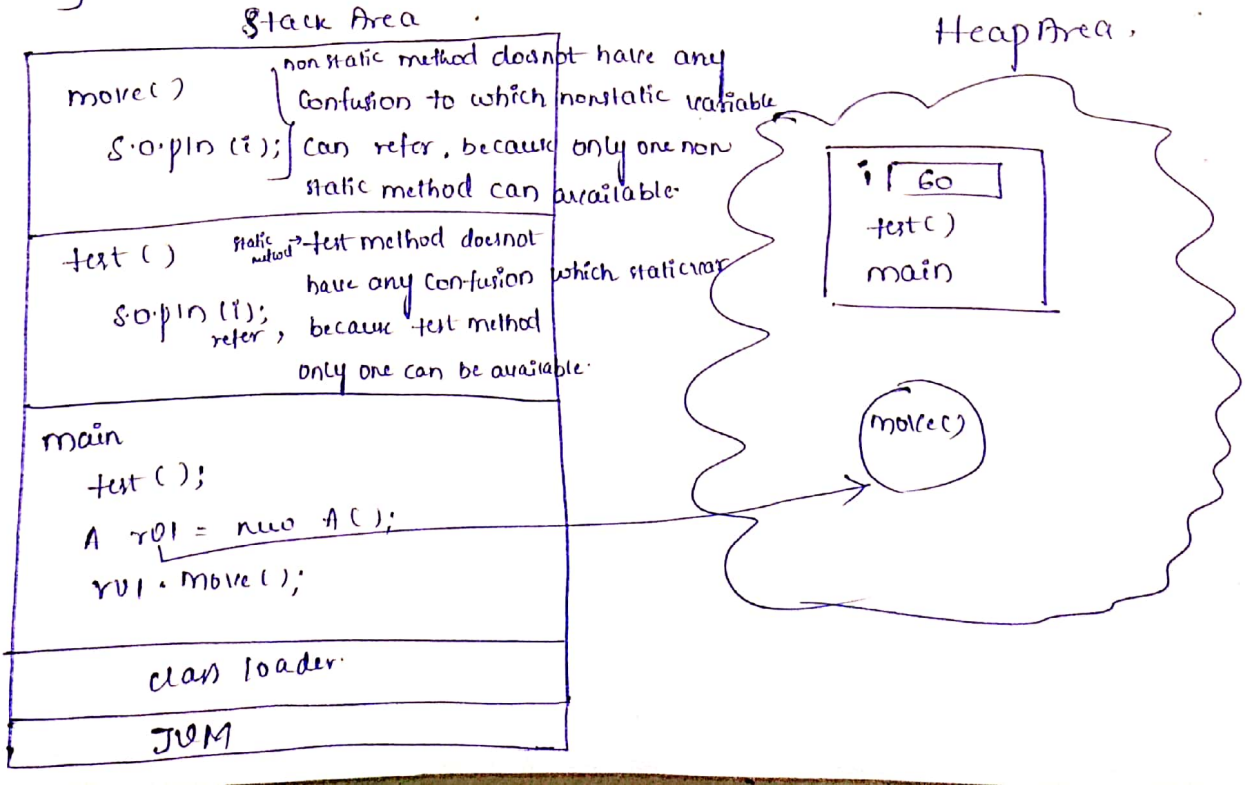
```
Class Demo
{
    static int i = 10;
    int j = 20;
    public static void main (String[] args)
    {
        S.o.pln (i);
        Demo r01 = new Demo();
        Demo r02 = new Demo();
        S.o.pln (r01.j);
        S.o.pln (r02.j);
    }
}
```



understanding accessing of datamembers within Member function :-

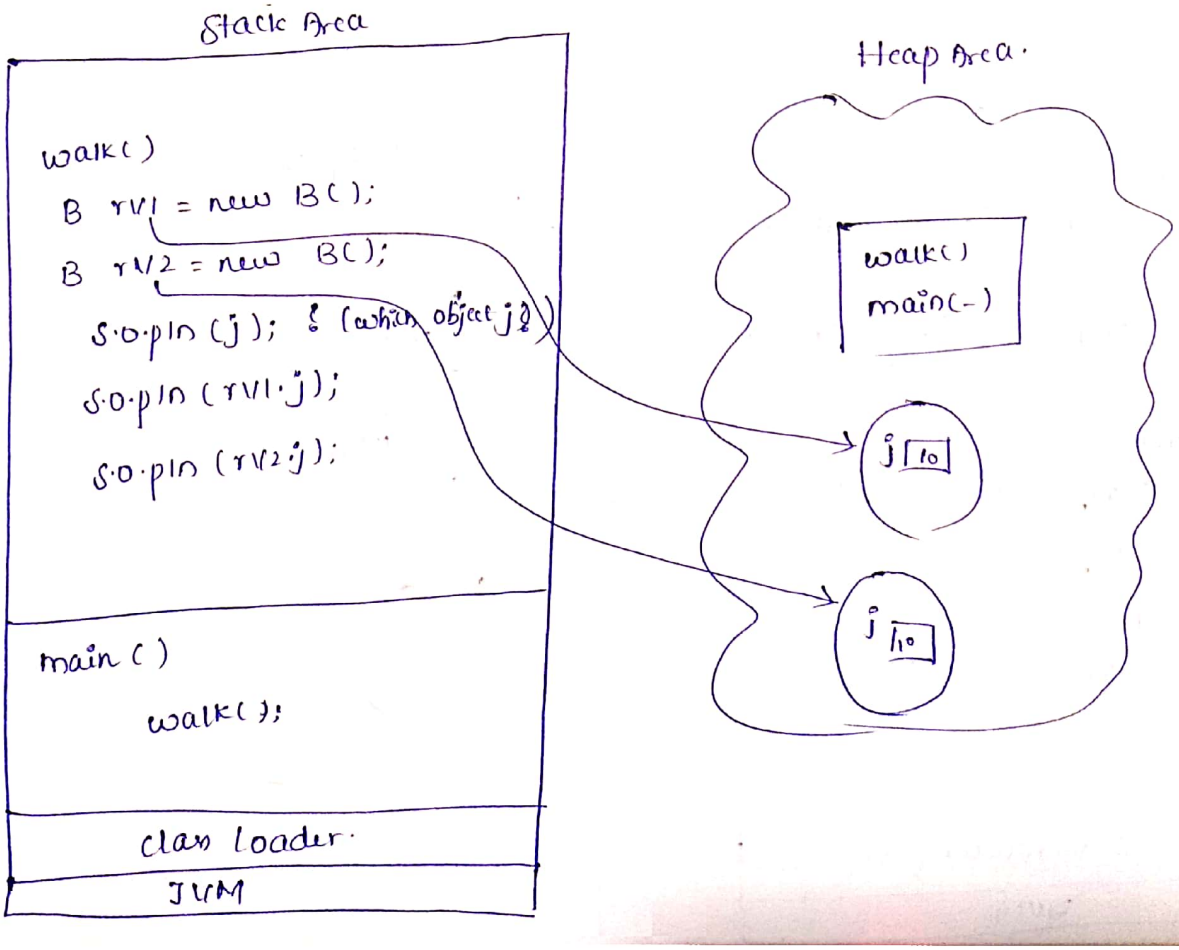
→ Static variable can be directly accessed within static method and nonstatic method because there is only one copy of static variable.

```
class A
{
    static int i = 60;
    public static void test():
    {
        S.O.pln (i);
    }
    public void move()
    {
        S.O.pln (i);
    }
    public static void main (String[] args)
    {
        test();
        A ro1 = new A();
        ro1.move();
    }
}
```



→ Nonstatic variable cannot be directly access from static method because there might be multiple copies of nonstatic variable. hence if static method refers nonstatic method directly it leads to Confusion. So, static method should access nonstatic variable through Objects address.

```
class B
{
  int j = 10;
  public static void walk()
  {
    B rv1 = new B();
    B rv2 = new B();
    S.O.pln(j);
    S.O.pln(rv1.j);
    S.O.pln(rv2.j);
  }
  public static void main(String[] args)
  {
    walk();
  }
}
```

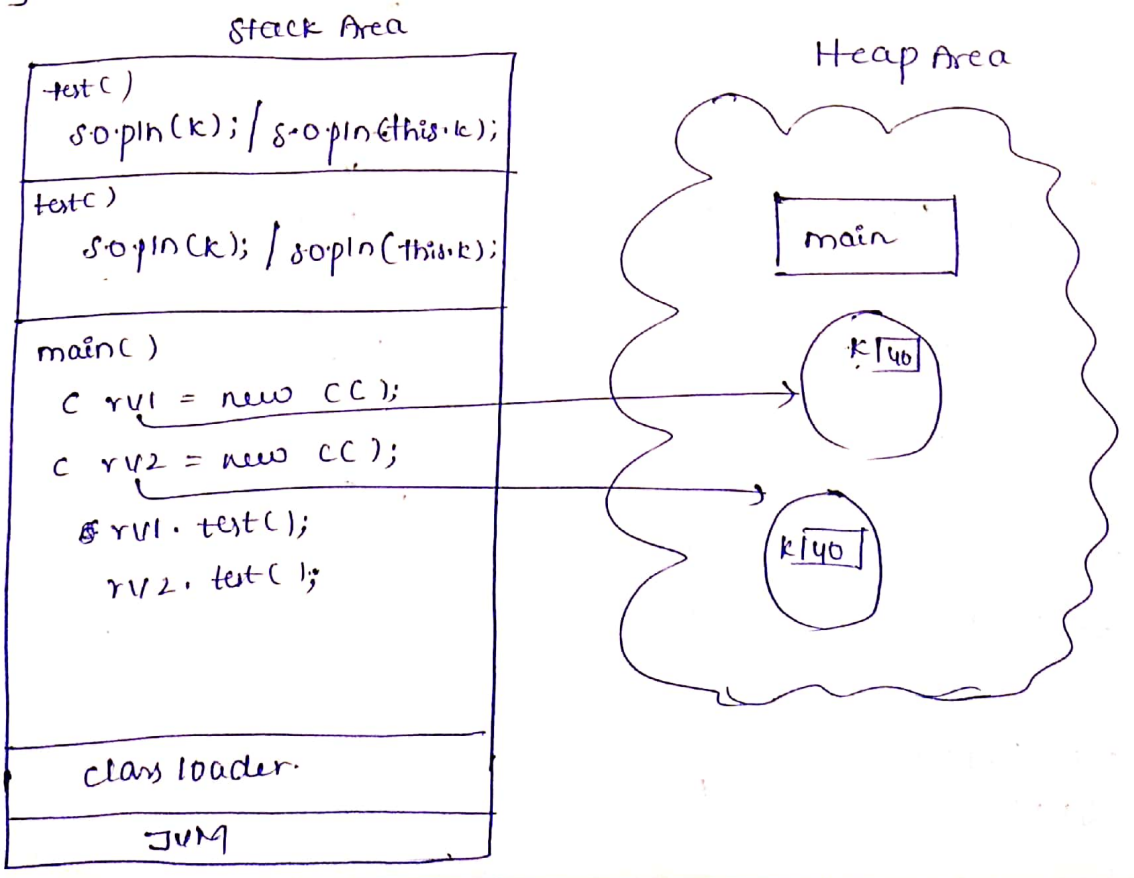


→ Nonstatic variable can be accessed directly from nonstatic method because every object will have a copy of nonstatic variable and nonstatic method. So, when nonstatic variable is referred directly from nonstatic method it always refers to current object nonstatic variable.

Note:-

Whenever we refer nonstatic variable within nonstatic method we a keyword "this" which represents current object in execution.

```
class C
{
    int k=40;
    public void test()
    {
        s.o.pln(k); / s.o.pln(this.k);
    }
    public static void main(String[] args)
    {
        C rv1 = new CC();
        C rv2 = new CC();
        rv1.test();
        rv2.test();
    }
}
```



11/03

Multiple Ways to access Static Members :-

→ There are three ways to access static members

(i) Directly

(ii) class name

(iii) object

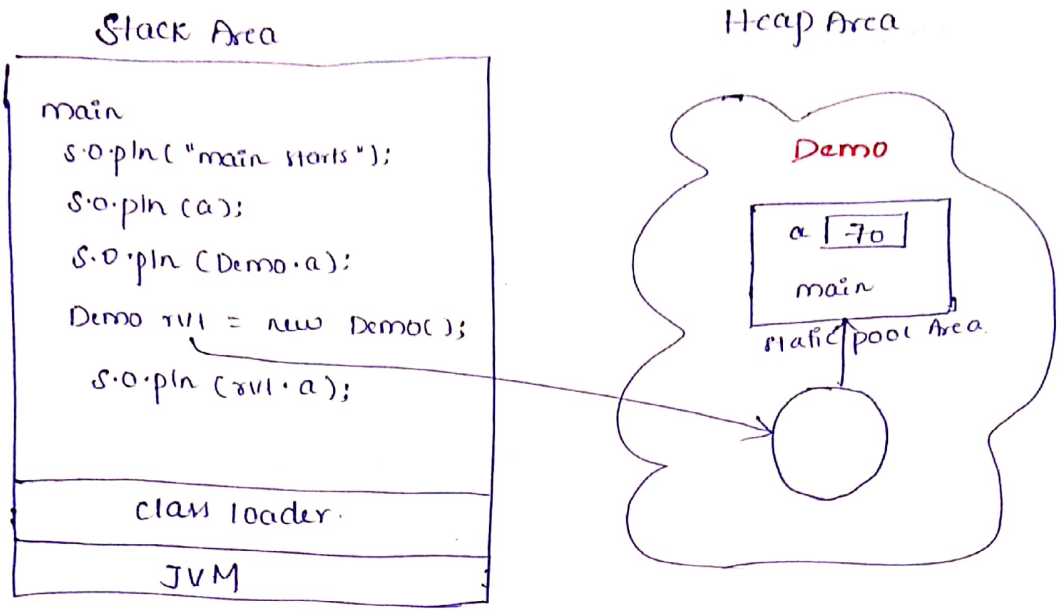
(i) Static members are access directly because there is only single copy.

(ii) Static members are access through class name because static pool Area is similar to classname.

(iii) Static member are access through object because every object is connected with static pool Area.

Prog:

```
class Demo
{
  static int a = 70;
  public static void main (String[] args)
  {
    S.O.pln ("main starts ....");
    S.O.pln (a);
    S.O.pln (Demo.a);
    Demo rv1 = new Demo();
    S.O.pln (rv1.a);
    S.O.pln ("main ends...");
  }
}
```



program 2 :— Instance variable & class variable.

```
class Run
{
   int a = 50;
   static int b=60;
   public static void main(String[] args)
   {
      s.o.pln("main starts...");
      Run r1 = new Run();
      Run r2 = new Run();
      s.o.pln(r1.a);
      s.o.pln(r2.a);
      s.o.pln(r1.b);
      s.o.pln(r2.b);
      r1.a = 55;
      r1.b = 99;
      r2.b = 120;
      s.o.pln(r1.a);
      s.o.pln(r2.a);
      s.o.pln(r1.b);
      s.o.pln(r2.b);
      s.o.pln("main ends...");
   }
}
```

Output :—

```
main starts
50
50
60
60
55
50
120
120
main ends
```

Output :—

```
main starts
50
50
60
60
55
50
120
120
main ends.
```

* Every object will have a copy of nonstatic variable hence nonstatic variables are also called as instance variables.
* Any changes made to nonstatic variable will effect only the current object.
* Entire class will have single copy of static variable hence static variables are also called as class variables.
* Any changes made to static variable, the changes will reflect to entire class.

12/03

Multiple classes :-

Sample.java.

```
class A
{
  public static void main(String[] args)
  {
    S.O.pln ("Inside A");
  }
}
class B
{
  public static void main(String[] args)
  {
    S.O.pln("Inside B");
  }
}
```

javac filename.java

A.class

javac Sample.java. javafilenam

A.class → java A

B.class → java B.

→ Single java file can have multiple classes
→ Each class can have its own main method.
→ It is not mandatory javafilename must be similar to classfilenam rather we can give any java filename.
→ Compiler will template generate dedicated classfile for each and every class.

→ During Compilation make use of JAVAFILENAME Ex: javac Sample.java

→ During Execution make use of "CLASS FILENAME" java A (A.class)

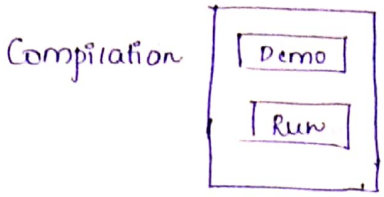
Accessing Members of one class in another class :-

Test.java

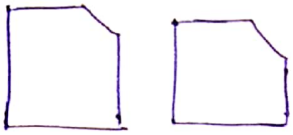
```
class Demo
{
    static int i = 40;
    int j = 30;
}
class Run
{
    public static void main (String[] args)
    {
        S.o.pln (Demo.i);
        Demo rvl = new Demo();
        S.o.pln (rvl.j);
    }
}
```

Summary

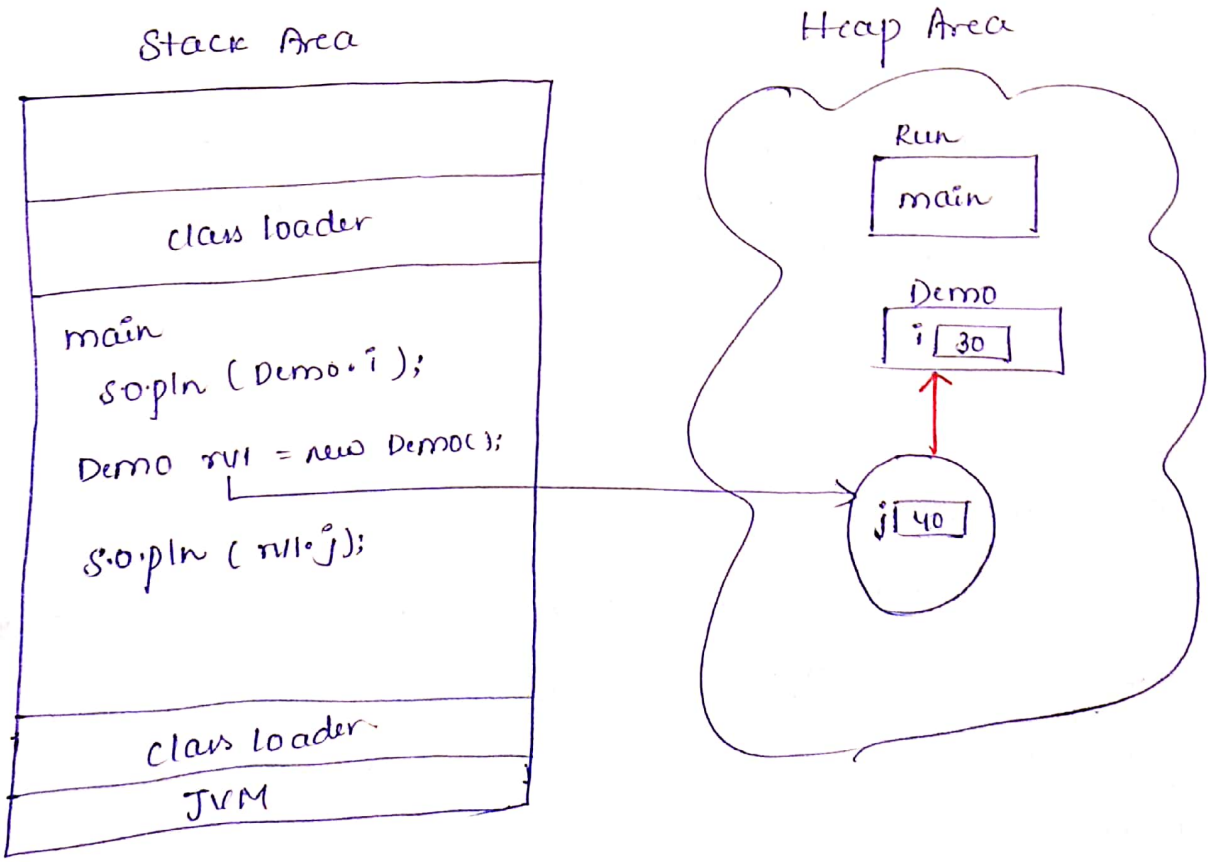
Test.java



javac Test.java



Execution :- java Run



→ Static members can be access in another class through classname.

→ Non static member can be accessed in another class through object creation.

→ For every class, class loader gets called whenever JVM encounter (seen) a new class for the first time.

→ For Every class static pool Area gets created.

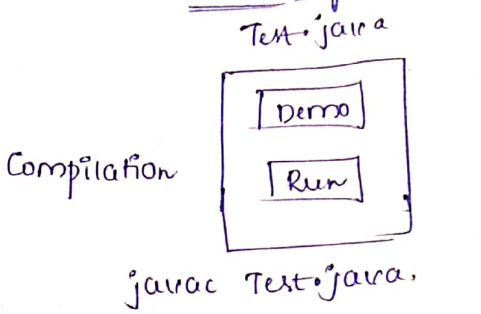
Proof :-

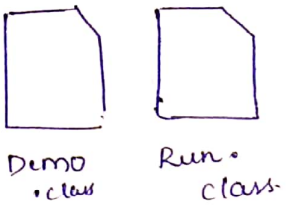
Proof for static members are loaded first before non static members

Test.java.

```
class Demo
{
  static int i = 30;
   int j = 40;
}
class Run
{
  public static void main(String[] args)
  {
    Demo rvl = new Demo();
    S.O.pln (rvl.j);
    S.O.pln Demo.i);
}
}
```

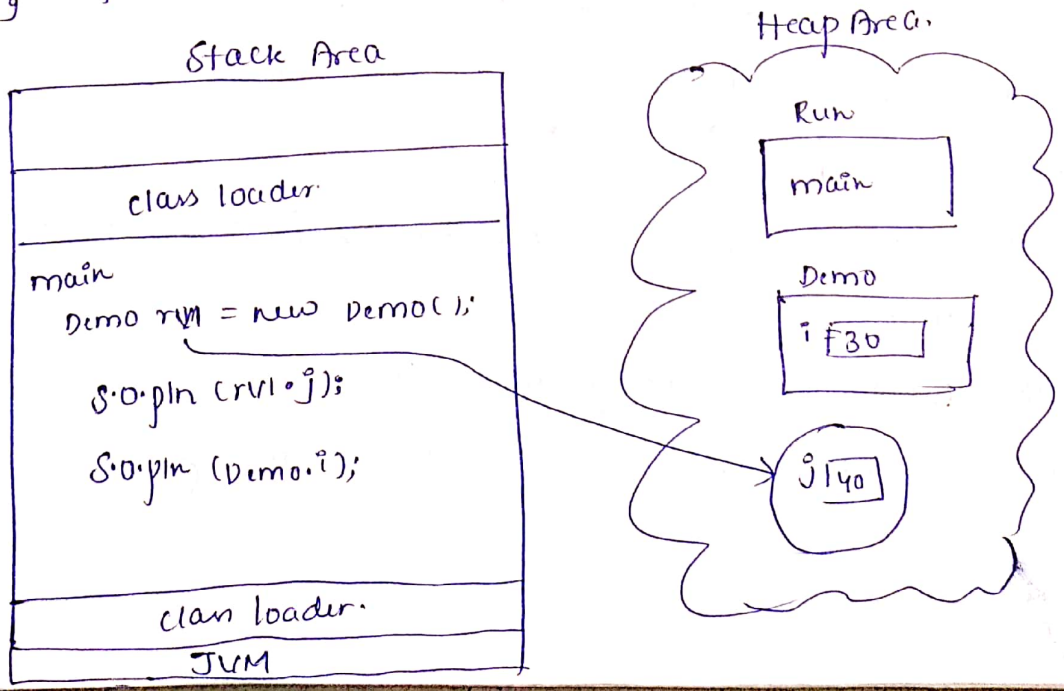
Summary.





Execution

java Run



`Demo rvl = new Demo();`.

In this program, the statement or syntax is for object creation JVM encounter new class for the first time it calls class loader. The class loader loads static members into static pool Area then comes back and creates object which loads non static members. Hence proved that static members are always loaded first. that is the reason why main method is loaded first.

13/03/19

CLASS AND OBJECT

* class is a "Blue print"
* Object is the "mirror image of class."
* class is "logical entities."
* object is "Real entities. (or) entity."

* Every Object will have state and behaviour.
* State represents "what object knows about itself"
* Behaviour represents "what object does"
* State is nothing but data. this data can be stored in a static variable and or nonstatic variable.
* If data changes dynamically for same variable, it is a nonstatic variable.
* If data remains same (doesnot changes) for a variable, it is a static variable.
* operatio Behaviour is nothing but operation, this operation can be defined in static method or non static method.

* If there is a dependency between state and behaviour, its a nonstatic method, then behaviour is a nonstatic method.
* If there is no dependency between state and behaviour, then behaviour is a static method.

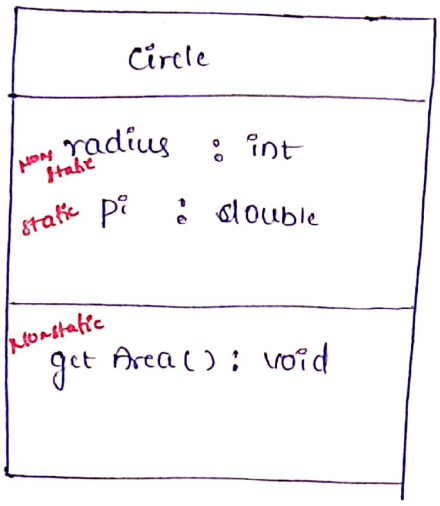
Note :-

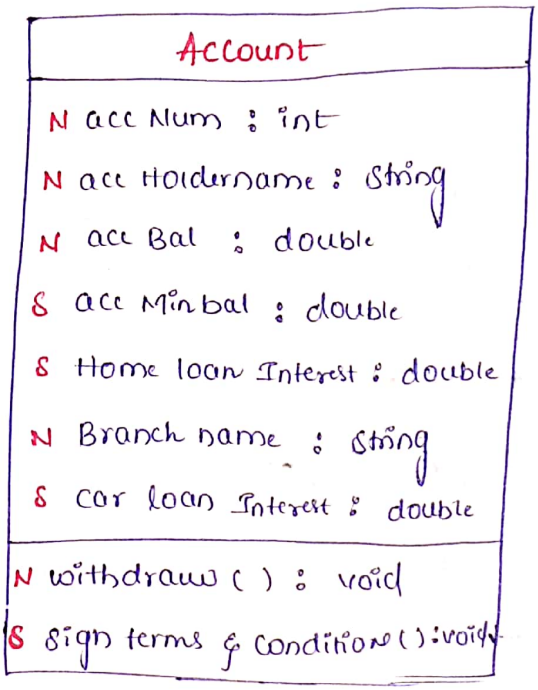
using one class we can create multiple objects.

Class diagram / UML Diagram :-

UML → unified Modelling Language.

Its a pictorial representation of a Java program.





S → static

N → Nonstatic

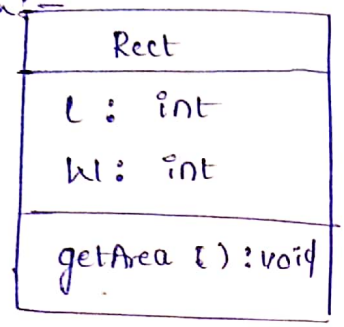
program :-

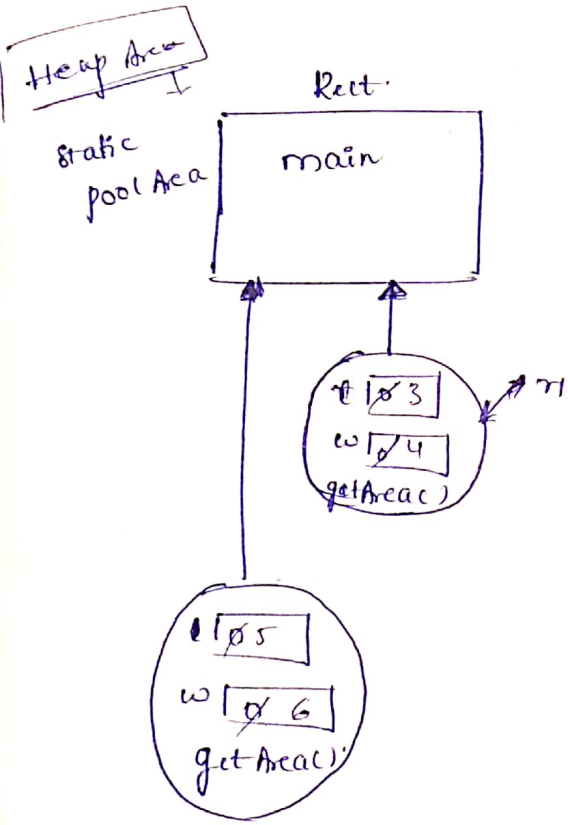
```
class circle
{
  int r;
  static double pi = 3.14;
  public void getArea().
  {
   double area = pi * this.r * this.r;
   S.O.pln (area);
  }
  public static void main (String[] args)
  {
     Circle c1 = new circle();
     Circle c2 = new circle();
```

```
        c1.r = 4;
        c2.r = 6;
        c1.getArea();
        c2.getArea();
    }
}
```

14/03/19

diagram:-





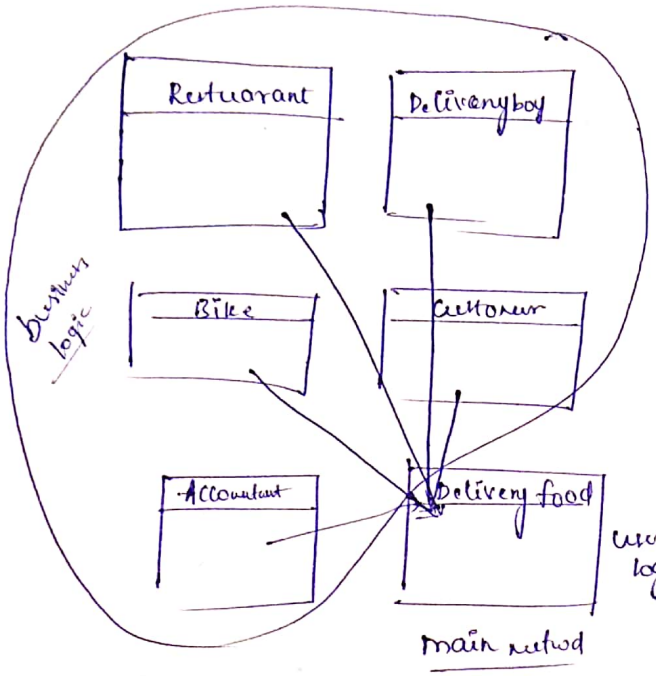
program:

```
class Rect
{
    int l;
    int w;
    public void getArea()
    {
        int area = this.l * this.w;
        S.o.pln(area);
    }
    public static void main (String[] args)
    {
        Rect r1 = new Rect();
        Rect r2 = new Rect();
        r1.l = 3;
        r1.w = 4;
        r2.l = 5;
        r2.w = 6;
        r1.getArea();
        r2.getArea();
    }
}
```

Application

Every application will have an business logic and user logic when the business logic process can be done by the business logic classes

For Example : online food order in foodpanda.



→ In this online food order application Restuarant is the buiness logic process. by using class it can be defined.

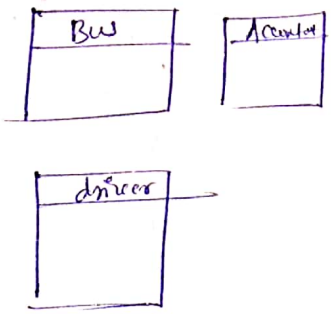
→ Delivery boy, bike, Customer and Accountant these all are the buiness logic class to process the Compilation.

→ When the delivery food Contains main method to execute all the classes to give the output. (-food).

user logic can be execute the program.

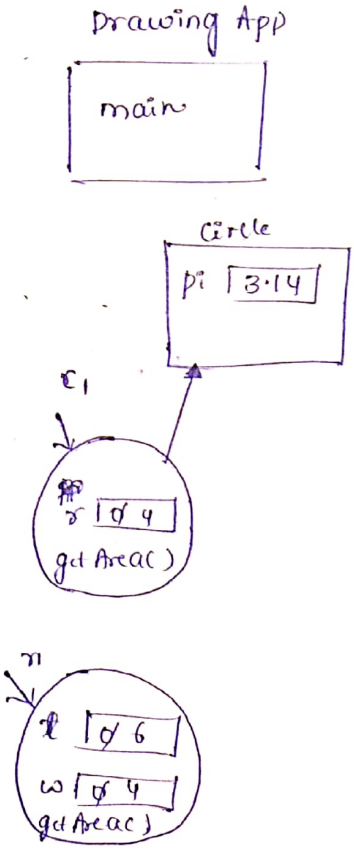
user logic

Example 2 :— Online bus tickets



Program :—

```
class Circle
{
    int r;
    static double pi = 3.14;
    public void getArea()
    {
      double area = pi * this.r * this.r;
      S.o.pln(area);
    }
}
class Rect
{
    int l;
    int w;
    public void getArea()
    {
      int area = this.l * this.w;
      S.o.pln(area);
    }
}
class DrawingApp
{
  public static void main(String[] args)
  {
    Circle c1 = new Circle();
    c1.r = 4;
    c1.getArea();
    Rect r1 = new Rect();
    r1.l = 6;
    r1.w = 4;
    r1.getArea();
  }
}
```



** 15/03/19

Note :-

According to standard practise Java file name will be similar to user's logic class name.

Current problem :-

* Non static variable initialization is compulsary by programmer because some of the methods makes use of nonstatic variable to perform the operation.

  For Examples getArea() method makes use of radius to calculate Area.

  For example Fire() method makes use of bullets to fire.

* In case programmer forgets to initialize nonstatic variable depending method will not operate properly.

* Even though we forgot to initialize nonstatic variable we will not get any error during compilation because nonstatic variables will be initialized with default values.

* Because of default values during execution methods are not operating properly. This mistake programmer realizes after execution.

* Realizing the mistake after mistake there is no use ~~of~~ execution

* It is better to realize the mistake during compilation. So that programmer will initialize nonstatic variable and methods will operate properly at the time of execution.

* Inorder to realize the mistake during compilation (forget to initialize the nonstatic variable) we must use "constructor".

* "Constructor will guarantee nonstatic variables are initialize by programmer."

Constructor :-

* Constructor is a type of method, whose name is similar to class name and gets executed each and everytime when object is created.
* Object creation and constructor call are Inter dependent.
* 'new' operator is responsible for calling the constructor.
* If programmer dosnot creates any constructor compiler will create default constructor.
* Constructor without any argument is called default constructor.

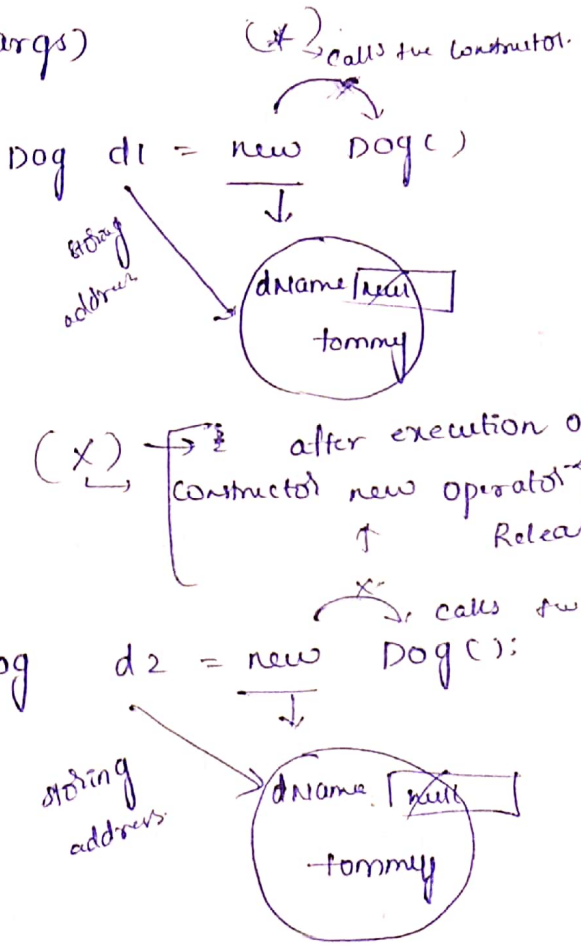
```
class Demo
{
    Demo()
    {
      S.O.pln(" In Demo class Constructor.... ");
    }
    public static void main(String[] args)
    {
      S.O.pln(" main Starts ");
      Demo rv1 = new Demo();
      Demo rv2 = new Demo();
      Demo rv3 = new Demo();
      S.O.pln(" main ends ... ");
    }
}
```

output :-

```
Main Starts
In Demo class constructor....
In Demo class Constructor....
In Demo class constructor...
Main ends.
```

Program :-

```
class Dog
{
    String dName;
    Dog()
    {
        this.dName = "tommy";
    }
}
class petShop
{
    public static void main (String[] args)
    {
        S.O.pln ("main starts");
        Dog d1 = new Dog();
        Dog d2 = new Dog();
        S.O.pln (d1.dName);
        S.O.pln (d2.dName);
        S.O.pln ("main ends");
    }
}
```



output :-

main starts

tommy

tommy.

main ends.

Constructor overloading :-

class Dog

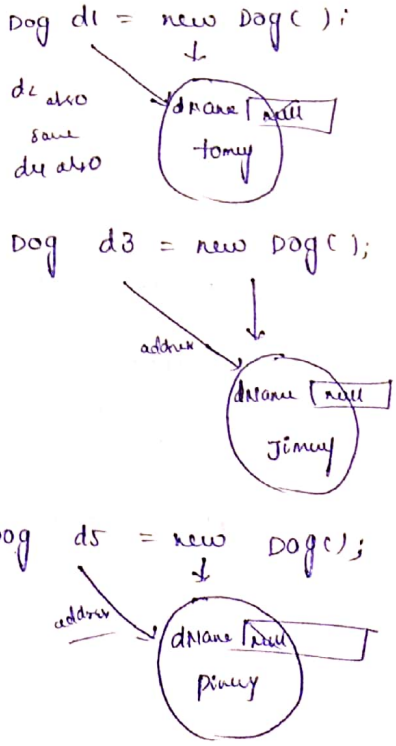
* Developing multiple constructors with different arguments is called constructor overloading.

Prog:

```
class Dog
{
    String dName;
    Dog()
    {
      this.dName = "tommy";
    }
    Dog(String dName)
    {
      this.dName = dName;
    }
}

class petshop
{
    public static void main(String[] args)
    {
        S.O.pln("main starts");
        Dog d1 = new Dog();
        Dog d2 = new Dog();
        Dog d3 = new Dog("Jimmy");
        Dog d4 = new Dog();
        Dog d5 = new Dog("pinky");
        S.O.pln(d1.dName);
        S.O.pln(d2.dName);
        S.O.pln(d3.dName);
        S.O.pln(d4.dName);
        S.O.pln(d5.dName);
        S.O.pln("main ends");
    }
}
```

Output:-

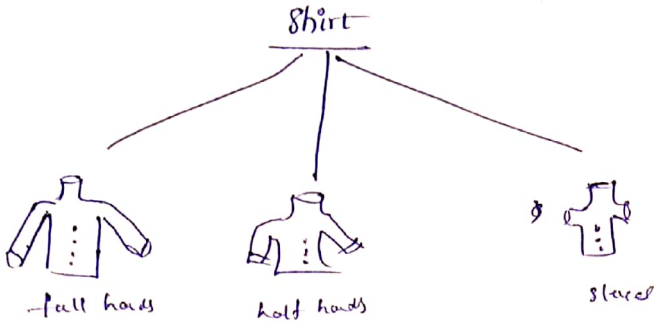
```
main starts
Tommy
Tommy
Jimmy
Tommy
pinky
main ends.
```



18/03/19

The main purpose of Constructor overloading is to create same class object in multiple waays.

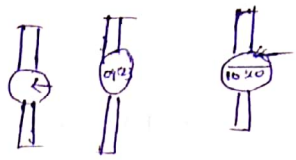
Realtime example :-



→ Shirt class object can be created in three ways

(i) full hands
(ii) Half hands
(iii) Sleeves.

watch



→ Watch class object can be created in three ways.

(i) Analog watch
(ii) Digital watch
(iii) both analog & digital

→ We can create multiple objects for same class

DRY principle :-

Write a program to store Information of car.

(i) car information such as brandname, top speed, mileage and color.

(ii) Some of the cars will have fixed information. (Hondacity, 140 km/hr, 9.7 km/l, Red)

(iii) Some of the cars will have fixed brandname and color (i20, white) but varying topspeed and mileage.

(iv) Some of the cars will have fixed topspeed and mileage (150 km/hr, 8.6 km/l) but varying brandname and color.

(v) Some of the cars will have varying brandname, topspeed, mileage and color.

(vi) cars are sold in cars showroom

print each and every car information

```
class car
{
    String bname;
    double tspeed;
    double mileage;
    String color;
    car()
    {
        this.bname = "Hondacity";
        this.tspeed = 140kmph;
        this.mileage = 9.7 kmpl;
        this.color = "Red";
    }
    car(String bname, String color)
    {
        this.bname = "I20";
        this.color = "White";
        this.tspeed = tspeed;
        this.mileage = mileage;
    }
    car(int tspeed, double mileage)
    {
        this.tspeed = 150 km/hr
        this.mileage = 8.6 km/l
    }
    car(String bname, String color)
    {
        this.bname = bname;
        this.color = color;
    }
```

Dry principle :—

car(int tspeed, double mileage) → car(int tspeed, double mileage)

car(String bname, String color)

```
car ( String bname , int tspeed , double mileage , String cold )
{
  this.bname   = bname ;
  this.tspeed  = tspeed ;
  this.doublmileage = mileage ;
  this.cold    = cold ;
}
```

```
public carInfo( )
{ S.O.pln (bname)
  "
  "
  "
}
```

```
class carshowroom
{
 public static void main ( String[] args)
 {
   S.O.pln (" main starts --- ");
   car c1 = new car( );
   car c2 = new car(120 , 6.5 );
   car c3 = new car("Audi", "white");
   car c4 = new car ("Benz" , "140 ", 5.7 , "Black");
   S.O.pln (" c1.bname, c1.tspeed , c1.mileage , c1.cold )
   S.O.pln ( c2.tspeed , c2.mileage);
   S.O.pln c3.bname , c3.cold );
   S.O.pln ( c4.bname , c4.tspeed , c4.mileage , c4.cold );
   S.O.pln (" main ends ... ");
 }
}
```

```
c1.carInfo();
c2  "
c3  "
c4  "
```

(i)

```
public void carInfo( )
{
  S.O.pln (this.bname);
  S.O.pln (this.tspeed);
  S.O.pln (this.mileage);
  S.O.pln (this.cold);
}
```

(ii):

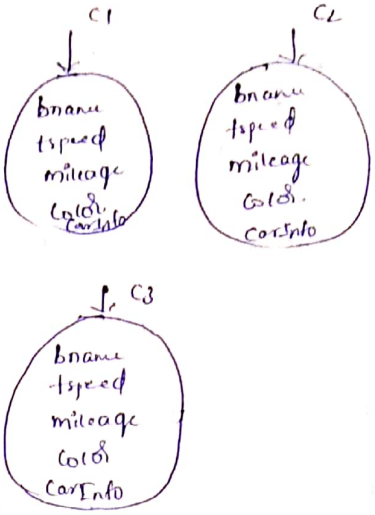
```
c1.carInfo( );
c2.carInfo( );
c3.carInfo( );
```

Program:- Dry principle:-

```
Class car
{
  String bname;
  String color;
  int tspeed;
  double mileage;
  car()
  {
    this.bname = "Hondacity";
    this.tspeed = 140;
    this.mileage = 9.7;
    this.color = "Red";
  }
  car(int tspeed, double mileage)
  {
    this.bname = "I20";
    this.color = "white";
    this.tspeed = tspeed;
    this.tspeed mileage = mileage;
  }
  car(String bname, String color)
  {
    this.bname = bname;
    this.color = color;
    this.tspeed = 150;
    this.mileage = 8.6;
  }
  car(String bname, String color, int tspeed, double mileage)
  {
    this.bname = bname;
    this.color = color;
    this.tspeed = tspeed;
```

```
    this.mileage = mileage;
}
public void carInfo()
{
    S.o.pln(bname);
    S.o.pln(tspeed);
    S.o.pln(mileage);
    S.o.pln(color);
}
}
```

Not written




```
class Showroom
{
    public static void main(String[] args)
    {
        S.o.pln("main starts ....");
        car c1 = new car();
        car c2 = new car(120, 6.5);
        car c3 = new car("I10", "white");
        car c4 = new car("Benz", 140, 5.7, "Black");

        S.o.pln(c1.bname);
        S.o.pln(c1.tspeed);
        S.o.pln(c1.mileage);
        S.o.pln(c1.color);

        S.o.pln(c2.bname);
        S.o.pln(c3.tspeed);
        S.o.pln(c2.mileage);
        S.o.pln(c3.color);
        S.o.pln(c3.bname);
        S.o.pln(c3.tspeed);
        S.o.pln(c3.mileage);
        S.o.pln(c3.color);
        S.o.pln(c4.bname);
        S.o.pln(c4.tspeed);
        S.o.pln(c4.mileage);
        S.o.pln(c4.color);
```

→ * Remove this and call the method

to c1 object by calling → c1.carInfo();

creating Method ←

```
public void carInfo()
{
    S.o.pln(bname);
    S.o.pln(tspeed);
    S.o.pln(mileage);
    S.o.pln(color);
}
```

19/03/19 :-

→ Above program consists of repeatative code, due to which we will face maintenance problem.

→ Maintenance problem means, if there is any code change in the future. The changes should be made where ever code is repeated.

→ So we must follow Dry principle

DRY → Donot Repeat yourself

→ Dry principle says convert repeatative code into reusable nonstatic method.

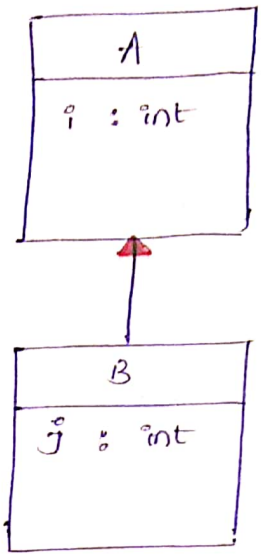
20/03/19

Inheritance :- [Is-A Relationship]

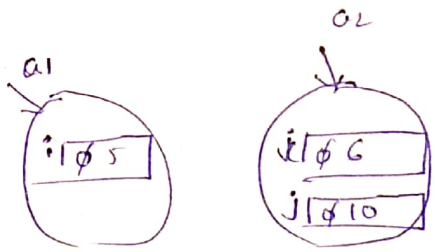
* Acquiring members of one class to another class is called Inheritance.
* Inheritance can be achieve by using extends keyword.
* using Superclass object we can access only Superclass Members.
* using Subclass object we can access both the Superclass members as well as subclass members
* only nonstatic members can be inherited; static members cannot be inherited. because there is only one copy of static members and it is already loaded by classloader.

Single level Inheritance :—

```
class A
{
  int i;
  A(int i)
  {
    this.i = i;
  }
}
class B extends A
{
  int j;
  B(int i, int j)
  {
    //this.i = i;  → Error  replace with super(i);
    this.j = j;   ↘ super(i);
  }
}
class Inprog1
{
  public static void main(String[] args)
  {
    A a1 = new A(5);
    S.o.pln(a1.i);
    B b1 = new B(6, 10);
    S.o.pln(b1.i);
    S.o.pln(b1.j);
  }
}
```

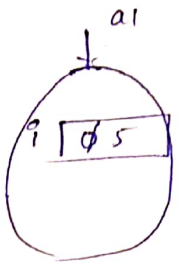


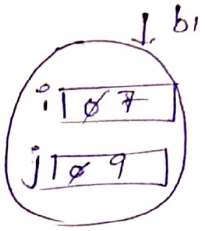
In inheritance program constructor is compulsory.



* In the above program is error because there is no constructor chaining between superclass and subclass.
* According to java syntax constructor chaining is compulsary during inheritance process.

```
class A
{
  int i;
  A(int i)
  {
    this.i = i;
  }
}
class B extends A
{
  int j;
  B(int i, int j)
  {
    super(i);
    this.j = j;
  }
}
class Inprog1
{
  public static void main(String[] args)
  {
    A a1 = new A(5);
    S.O.pln(a1.i);
    B b1 = new B(7,9);
    S.O.pln(b1.i);
    S.O.pln(b1.j);
  }
}
```





→ call to super i.e. call to superclass Constructor.

Output:
5
7
9

In the above programs constructor chaining is happening in explicit way. i.e., if super class contains constructor with arguments, then its programmers responsibility to define call to super statement with appropriate argument in sub class constructor.

** ow
- explicit means → we can define arguments and super call to super then it is called
- implicit means → compiler adds the default gives args and call to super without any argument.

21/03/19

```
class E
{
    EC)
    {
        S.O.pln("E class constructor");
    }
}
class F extends E
{
    FC)
    {
        super();   → Compiler adds.
        S.O.pln("F class constructor");
    }
}
class Inherprg3
{
    public static void main(String[] args)
    {
        F f1 = new F();
    }
}
```

output :-

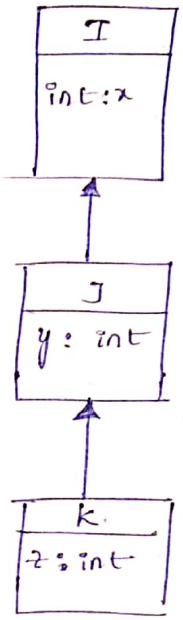
E class constructor
F class constructor.

Interview question :-

```
class G
{   GC){}   } compiler adds expli implicitly.
}

class H extends G.
{
    HC)
    { super(); }   } compiler adds implicitly.
}
```

In the above program Constructor chaining is implicit i.e., Super class contains default constructor. compiler will add call to super statement without argument in subclass Constructor.

Multi-level Inheritance :-

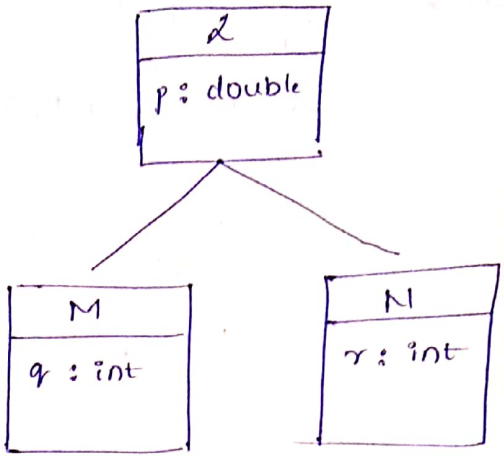



```
class I
{
  int x;
  I(int x)
  {
    this.x = x;
  }
}
class J extends I
{
  int y;
  J(int x, int y)
  {
    super(x);
    this.y = y;
  }
}
class K extends J
{
  int z;
  K(int x, int y, int z)
  {
    super(x, y);
    this.z = z;
  }
}
class Multiprog1
{
  public static void main(String[] args)
  {
    I o1 = new I(5);
    S.o.pln(o1.x);
    J o2 = new J(5,10);
    S.o.pln(o2.x);
    S.o.pln(o2.y);
    K o3 = new K(5, 10, 15);
    S.o.pln(o3.x);
    S.o.pln(o3.y);
    S.o.pln(o3.z);
  }
}
```

Output :-

```
5
5
10
5
10
15
```

Hierarichal Inheritance :-




```
class A
{
  double p;
  A(double p)
  {
    this.p = p;
  }
}
class M extends A.
{
  int q;
  M(double p, int q)
  {
    super(p);
    this.q = q;
  }
}
class N extends A.
{
  int r;
  N(double p, int q, int r)
  {
   super(p);
   this.r = r;
  }
}
class Hierarichal
{
  public static void main(String[] args)
  {
    A o1 = new A(5.5);
    so.pln(o1.p);
    M o2 = new M(5.5, 5);
    so.pln(o1.p);
    so.pln(o1.q);
    N o3 = new N(5.5, 10)
    so.pln(o3.p);
    so.pln(o3.r);
  }
}
```

→ Single superclass can have multiple subclass

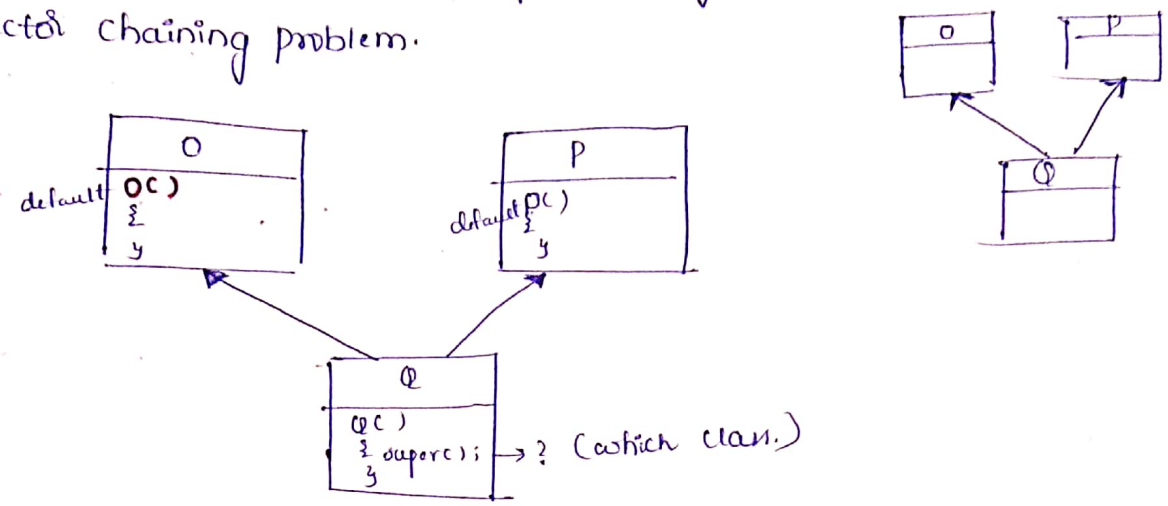
output :-

```
5.5
5.5
5
5.5
10
```

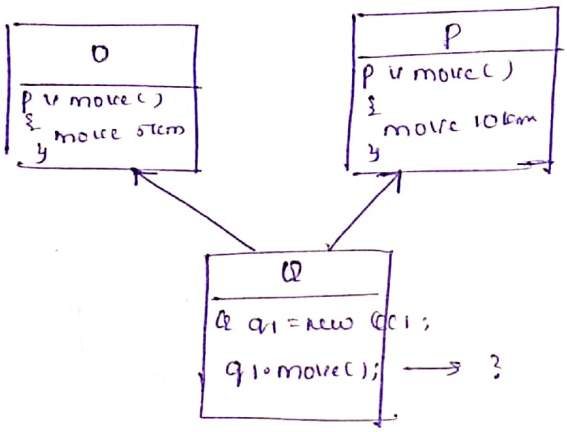
22/03/19

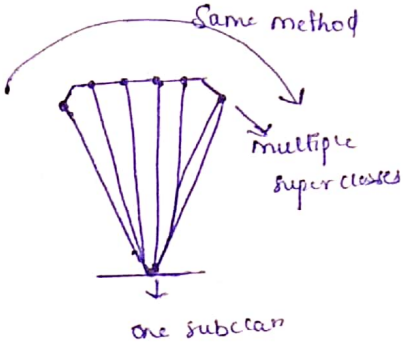
Multiple Inheritance :-

→ Multiple inheritance is not possible by using classes because of constructor chaining problem.



"Diamond problem" :-



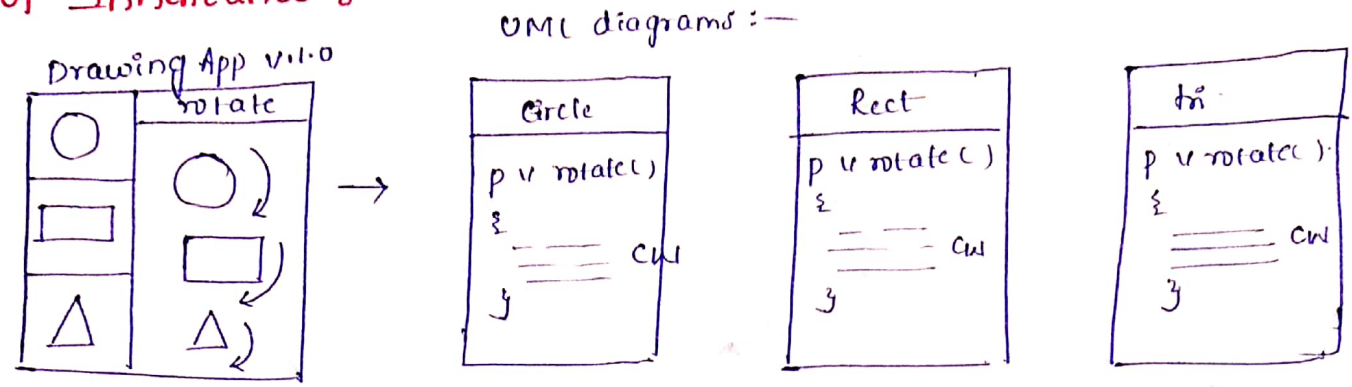


→ If subclass tries to inherit from multiple super classes, incase super classes have same method, ∴ subclass gets confused "which super class method to call." This is called as diamond problem.

** ambiguity → confusion
- which superclass method to call
- which superclass constructor to chaining.

→ Both the problems represent ambiguity.
(i) which superclass method to call
(ii) which superclass constructor to chain.

Use of Inheritance :—



* In the above design there are repeatative codes due to which we will face maintenance problems.

* We can avoid repeatative code and make use of reusability concept with the help of Inheritance. i.e., if multiple classes demands same code (circle demands rotating clockwise, rectangle demands rotating clockwise and triangle also demands rotating clockwise). instead of repeating the same code in each and every classes, define the code in single class (superclass) and reuse in all other classes. (subclass).

Note 1 :—

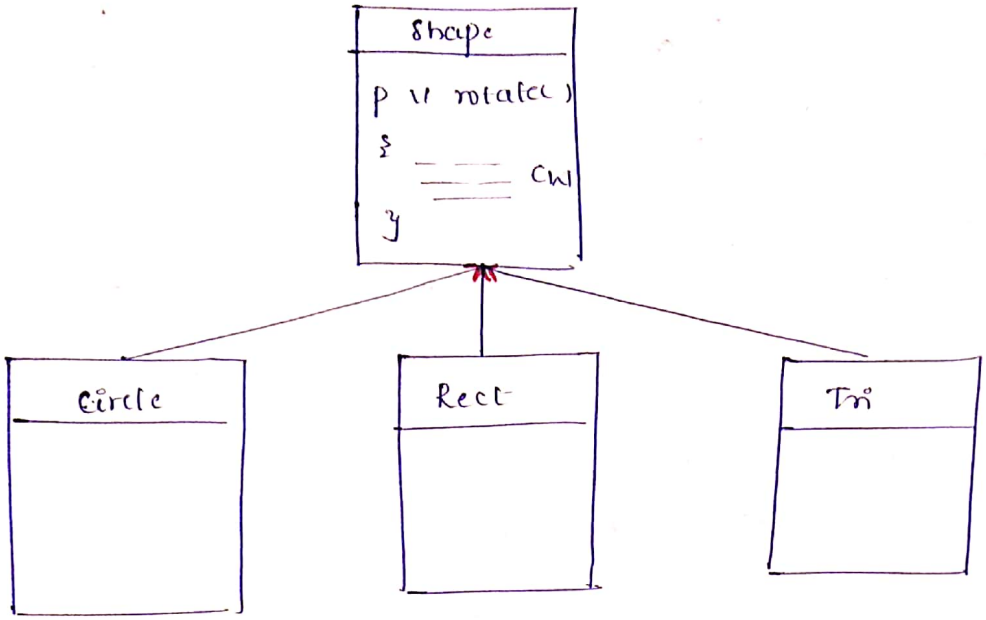
If the feature is common (feature can be a variable & method) define the feature in superclass and reuse it in subclasses.

Note 2 :—

If the feature is specific, define the feature in subclass and use it in the same subclass. ↳ belongs to present class

Modify design :—

Modify design :-



Circle Is-A Type of shape

Rect Is-A Type of shape.

Tri Is-A Type of shape.

Method overriding :-

→ During Inheritance process, Subclass can change method implementation of inherited method. This process is called Method overriding.

→ Method overriding depends on two factors.

(i) Inheritance is compulsary.

(ii) Subclass should maintain same method signature depending Superclass.

program :-

```
class A
 {
    public void walk()
     {
       S.O.pln (" Take a walk in mrng....");
     }
}
class B extends A
 {
    public void walk()
     {
       S.O.pln (" Take a walk on beach...");
     }
}
class Inhrprg
{
   public static void main (String[] args)
   {
     S.O.pln (" main starts... ");
     B b1 = new B();
     b1.walk();
     S.O.pln (" main ends.... ");
   }
}
```

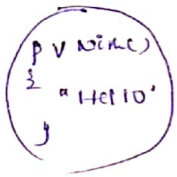
output :-

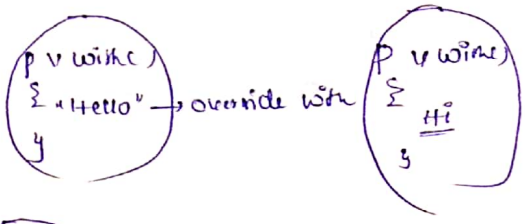
```
main starts....
Take a walk on beach....
main ends....
```

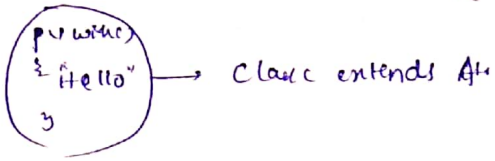
25/03/19:-

program for Method overriding :-

```
class A
{
  public void wish()
    {
      S.O.pln("Hello");
    }
}
class B extends A
  {
    public void wish()
      {
        S.O.pln("Hi");
      }
  }
class C extends A
  {

  }
class MethodOverriding
{
  public static void main(String[] args)
    {
      A a1 = new A();
      a1.wish();
      B b1 = new B();
        b1.wish();
      C c1 = new C();
        c1.wish();
    }
}
```





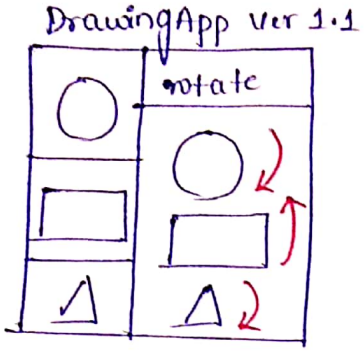


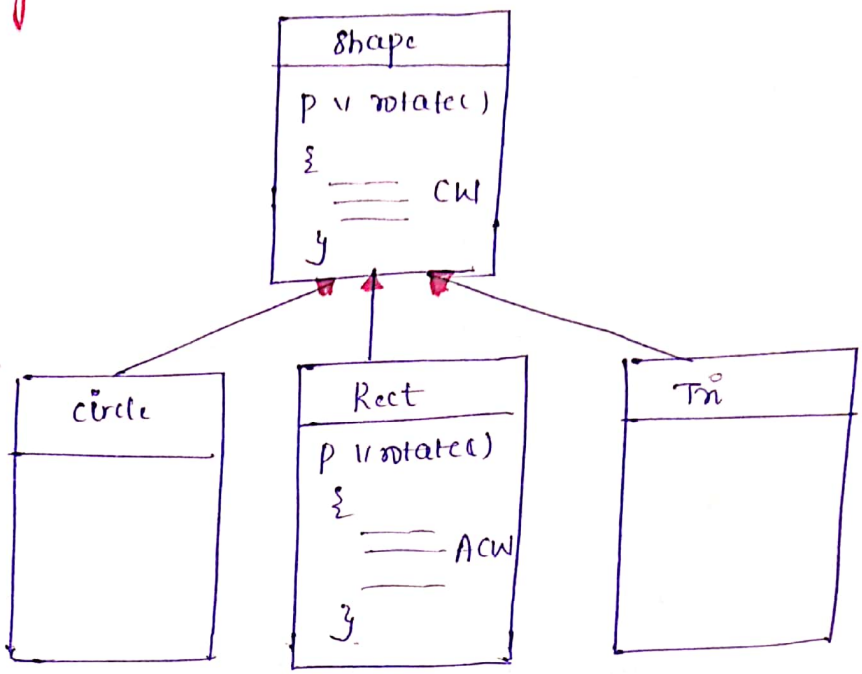
output :-

Hello
Hi
Hello.

→ If a subclass overrides the method, it effects only that particular subclass.

Use of Method overriding :-

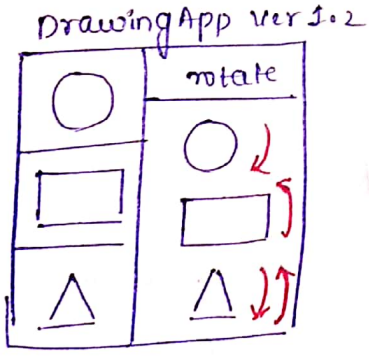


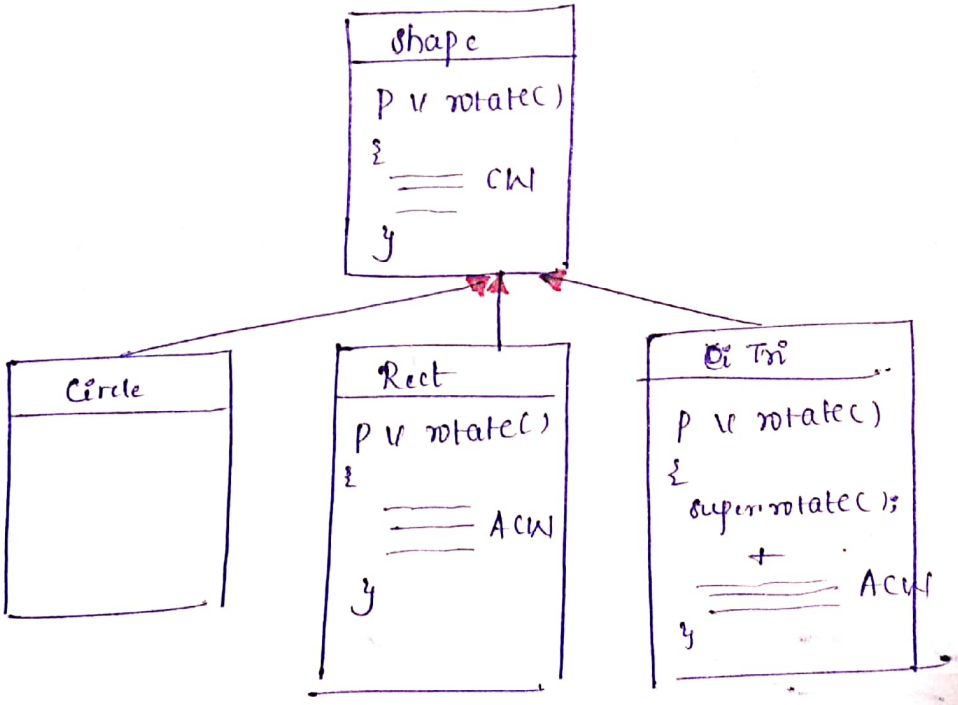


Realtime example :- Contacts, whatsapp, ringtones.

Whatsapp → suppose ur blocking one person it affects only that particular person not all the members.

→ If we need to change method implementation of inherited method for a particular subclass without affecting other subclasses. We go for Method overriding.

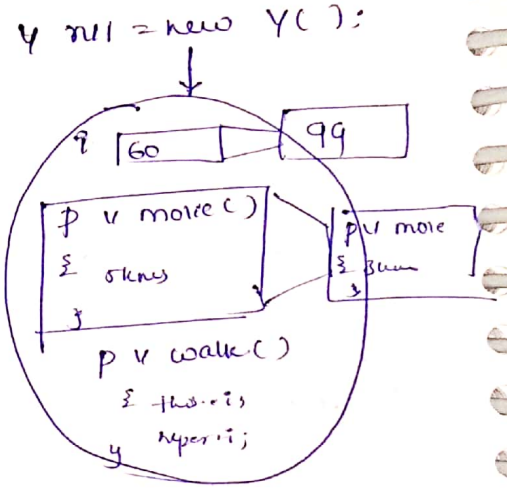




→ using super statement we can reuse superclass code which exists inside subclass object.

Program :-

```
class X
{   int i = 60;
    public void move()
    {
        S.O.pln(" move 5kms");
    }
}
class Y extends X.
{
    int i = 99;
    public void walk();
    {
        S.O.pln(this.i);
        S.O.pln(super.i);
    }
    public void move();
    {
        S.O.pln(" move 3 kms");
        super.move();
    }
}
class Demo
{
    public static void main(String[] args):
    {
        S.O.pln(" mains starts...");
        Y r1 = new Y();
        r1.walk();
        r1.move();
    }
}
```



mains starts
99
60
3 kms
5 kms.

Final Keyword :—

Final variable :—

→ Final variables cannot be reinitialized

→ Final variables are called as constant variables.

→ Constant variable can be created at three levels in Java.

Class level : Declare static variable as final

Object level : Declare nonstatic variable as final.

Method level : Declare local variable as final.

```
class Test
{
  public static void main(String[] args)
  {
     S.o.ph("main starts");
     final int i = 89;
     i = 78;  ⟶ Error because final variable cannot be reinitialized
     S.o.pln(i);
     s.o.pln("main ends");
  }
}
```

output: 89

26/03/19

Final Method :—

→ If method is declared as final those methods can be inherited and used. but cannot be overrided.

Program :—

```
class A
{
final public void punch()
  {
     s.o.pln("punch on face");
  }
}
```

```
class B extends A
{  [ public void punch() {
   }                    so.pln("punch on head"); ]
class MethodOverridingprog
{
   public static void main(String[] args)
   {
      S.O.pln("main starts");
      B rvl = new B();
      rvl.punch();
      S.O.pln("main ends");
   }
}
```

→ Error.
Super class method is final method then the subclass method is cannot be override.

Output :- Error.

Final class :-

→ If class is declared as final those classes cannot be inherited further.

→ final class can have super class but cannot have subclass.

→ no class can become subclass to final class.

→ final class is also called as last class in Inheritance hierarchy.

Note :-
final classes can we used in another class without inheritance.

Program :-

```
class M
{
}
class N extends
final class N extends M
{
}
[ class O extends N
  {
  } ]
```

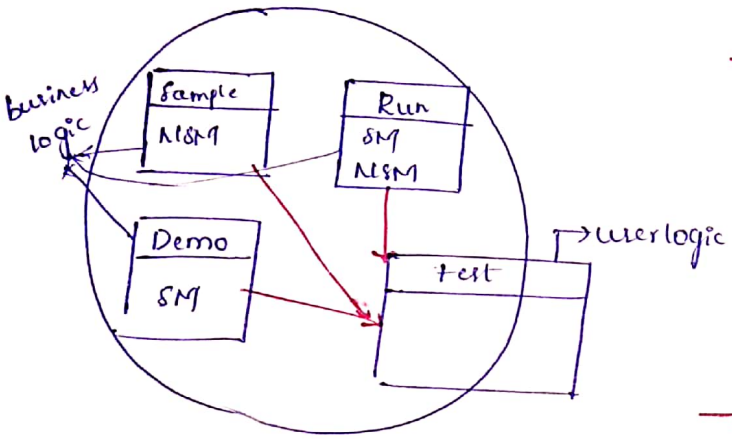
Error :- because N class is final O class cannot become a subclass.

Abstract :-

→ If class is declared as abstract, instantiation is not possible (object creation) is not ponible.

→ class is declared as abstract in three cases.

Case (i): class contains only static members, these members can be access through class name. It is unnecessary to create object and access static members. so block the object creation by declaring class as abstract.



→ When the program contains 3 buisness logic codes. then the each class contains Members and then the classes names are Sample, Demo and Run.

→ and this program contains userlogic code also then the class name can be declared as Test.

→ When the Sample class contains only Nonstatic members. These non static members can be declared nonstatic variable and non static methods.

→ When the Demo class contains only static Members. These static members can be declared as static variable and static method.

→ When the Run class contains both the static and nonstatic members where as static members can be declared as static variables and static methods and non static members can be declared as nonstatic variab and nonstatic methods.

program :—

Sample.java

```
class Sample
{
    int a = 50;
    public void travel()
    {
        S.o.pln("Travel to north india");
    }
}
```

Demo.java

```
abstract class Demo
{
    static int b = 55;
    public static void drive():
    {
        S.o.pln("Drive by wing bike");
    }
}
```

Run.java

```
class Run
{
    static double d = 5.4;
    int x = 60;
    public static void bike()
    {
        S.o.pln("Take an bike method");
    }
    public void slide()
    {
        S.o.pln("slide in water");
    }
}
```

Test.java

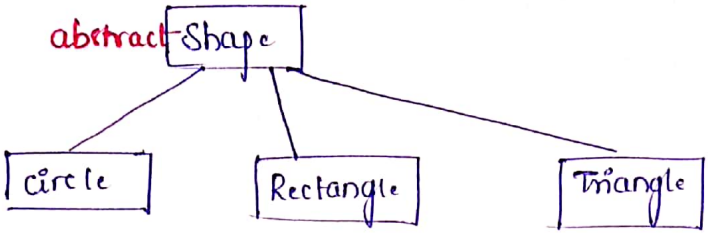
```
class Test
{
    public static void main (String[] args)
    {
        S.O.pln ("main starts");
        S.O.pln ("_ _ _ _ _ _ _ _ _");
        Sample s1 = new Sample( )
        S.O.pln (s1.a);
        S.O s1.travel( );
        S.O.pln( "_ _ _ _ _ _ _ _ _");
        [ Demo d1 = new Demo( ); ]
        S.O.pln (d1.b);
        d1.drive( );
        S.O.pln ("_ _ _ _ _ _ _ _");
        S.O.pln (Run.d);
        Run.hike( );
        Run r1 = new Run ( );
        S.O.pln (r1.x);
        r1.slide( );
        S.O.pln ("_ _ _ _ _ _ _ _");
        S.O.pln ("main ends");
    }
}
```

Error :- Demo class can be declared as abstract class . abstract class cannot be creating an object.

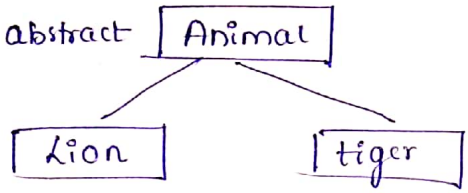
8 case (ii) :-

Superclasses are made as abstract because superclass objects doesnot exist in the real world. purpose of superclass is to hold common properties which are not used for superclass rather used for subclasses.

Ex:



Ex:



→ Circle, Rectangle, Triangle these are real word entities in the world so we have to create an object

→ Shape does not exist in real world entity so we can block the object and make the superclass as abstract class.

27/08/19

Normal Method :- Complete Method
Overriding is optional

Final Method :- Complete Method
cannot be override

Abstract Method :- Incomplete method
It can be overrided

28/03/19

Case(iii):-

→ If class contains abstract method, declare class as abstract.

x [In the class we can use abstract at the time object creation is blocked.] x

If we didnot mention the abstract class some other developers may create object and call abstract method which doesn't have method implementation. At that time JVM will face problems to avoid that make class as abstract class abstract keyword block the object creation.

→ Abstract methods will have only method signature, it should end with (;) and method should be declared as abstract.

→ Abstract methods are called Incomplete methods.

→ Abstract methods are called unimplemented methods.

→ Abstract methods can be implemented by subclass / incompleted by subclass / override by subclass.

→ Abstract methods specifies subclass should implement the method and when implemented subclass should maintain standard method signature. This is called "Standardization and specification."

Mini project :- Drawing App

Circle	○	1. rotate — CW 2. getArea — πr^2 3. fillColor — Red 4. getCircumference — $2\pi r$
Rect	▭	1. rotate — CW 2. getArea — L * W 3. fillColor → green 4. getPerimeter —
Tri	◺	1. rotate — CW 2. getArea — $1/2 \times b \times h$ 3. fillColor — black
	◺	1. rotate — CW 2. getArea — $1/2 bh$ 3. fillColor — blue
	△	1. rotate — CW 2. getArea — $1/2 bh$ 3. fillColor — Yellow

Program :-

Shape.java

```
abstract class Shape
{
    public void rotate()
    {
        S.O.pln("Rotate in clock-wise direction");
    }
    abstract public void getArea();
    abstract public void fillColor();
}
```

Circle.java

```
class Circle extends Shape
{
final static double pi = 3.14;
    int r;
    Circle(int r)
    {
        this.r = r;
    }
    public void getArea()
    {
        S.O.pln("Area of the circle is "+(pi * this.r * this.r));
    }
    public void fillColor()
    {
        S.O.pln("Circle color is Red");
    }
}
```

```
class Rect extends Shape
{
    final int l;
    int w;
    Rect(int l, int w)
    {
    }
    public void getCircumference()
    {
        S.O.pln("Circumference of the
                Circle is "+(2 * pi * this.r));
    }
}
```

Rect.java

```
class Rect extends Shape
{
    int l;
    int w;
    Rect(int l, int w){
    {
        this.l = l;
        this.w = w;
    }
    public void getArea()
    {
        S.O.pln("Area of the Rect is "+(this.l * this.w));
    }
    public void fillColor()
    {
        S.O.pln("Rect Color is green");
    }
    public void getperimeter()
    {
        S.O.pln("perimeter of the Rect is "+(2*(this.l + this.w)));
    }
}
```

Tri.java

```
abstract class Tri extends Shape
{
    int b;
    int h;
    Tri(int b, int h)
    {
        this.b = b;
        this.h = h;
    }
    public void getArea()
    {
        SO.pln("Area of the Tri is "+(0.5 * this.b * this.h));
    }
}
```

```
class RightAngleTri extends Tri
{
  RightAngleTri (int b, int h)
   {
     super(b,h);
   }
  public void fillColor ()
   {
     S.O.pln (" Right Angle Tri Color is blacle ");
   }
}

class LeftAngleTri extends Tri
 {
    LeftAngleTri (int b, int h)
    {
      Super (b, h);
    }
    public void fillColor ()
     {
       S.O.pln (" Left Angle Tri Color is blue ");
     }
 }

class EquilateralTri extends Tri
 {
   EquilateralTri (int b, int h)
    {  super(b,h);
    }
    public void fillColor()
    {
      S.O.pln (" Equilateral Tri Color is Yellow");
    }
}
```

finish the program by defining user's logic Drawing App define main method create each and every buisness logic program Object invoke all the methods.

```
Class Miniproject
{
  public static void main (String[] args)
  {
    S.O.pln ("main starts ...");
    Circle c1 = new circle (6);
    c1.rotate ();
    c1.getArea ();
    c1.getCircumference ();
    c1.fillColor ();
    S.O.pln ("----------");
    Rect r1 = new Rect (3,4);
    r1.rotate ();
    r1.getArea ();
    r1.getperimeter ();
    r1.fillColor ();
    S.O.pln ("----------");
    RightAngleTri rt1 = new RightAngleTri (5,10);
    rt1.rotate ();
    rt1.getArea ();
    rt1.fillColor ();
    S.O.pln ("----------");
    LeftAngle Tri lt1 = new LeftAngle Tri (6,7);
    Lt1.rotate ();
```

```
tl.getArea();
tl.fillcolor();
S.O.pln("---------");
EquilateralTri et1 = new EquilateralTri(7,11);
et1.rotate();
et1.getArea();
et1.fillcolor();
S.O.pln("---------");
}
}
```

* If a class inherits an abstract method, class will have 2 choices.

Choice :1 :- Abstract method must be overrided (or)

Choice -2 :- Declare class as abstract.

Note :- Abstract method can be overided anylevel of subclass.

* Abstract static methods cannot be develop because according to abstract, method must be inherited and overrided but at the same time according to static, method cannot be inherited and overrided.

[abstract public static void tl();] ⟶ not ponible

[abstract public void tl();] ⟶ possible

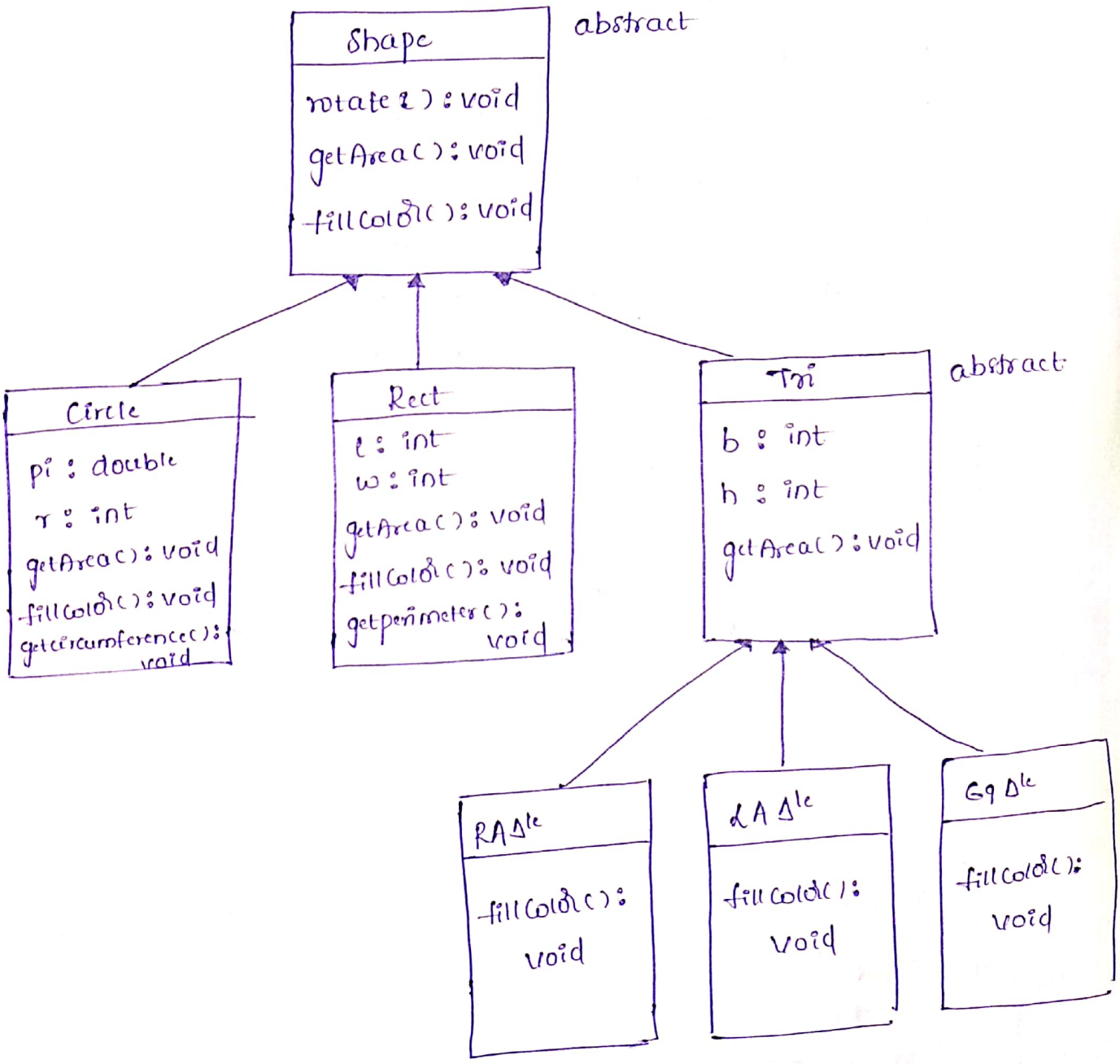
* abstract methods cannot be made as final because according to abstract, method must be Inherited and overrided but at the same time according to final, methods can be inherited but cannot be overrided.

[final abstract public void m1();] → Not possible.

10/04/19

Continous of Miniproject :-

UML diagram of Miniproject :-



Level - 2 :-

Modify Version of drawing application program :-

```
class DrawingApp
{
  public static void main (String[] args)
  {
    allBeh (new Circle(5));
    allBeh (new Circle(7));
    allBeh (new Rect(4,3));
    allBeh (new Rect(2,5));
    allBeh (new RightAngleTri (5,6));
    allBeh (new RightAngleTri (7,3));
    allBeh (new LeftAngleTri (5,3));
    allBeh (new LeftAngleTri (1, 5));
    allBeh (new EquilateralTri (5,7));
    allBeh (new EquilateralTri (1,3));
  }
  public static void allBeh (Circle c1).
  {
    c1.rotate();
    c1.getArea();
    c1.getCircumference();
    c1.fillColor();
  }
  public static void allBeh (Rect r1)
  {
    r1.rotate();
    r1.getArea();
    r1.fillColor();
    r1.getperimeter();
  }
```

```
public static void allBeh (RightAngleTri rt1)
{
    rt1.rotate();
    rt1.getArea();
    rt1.fillColor();
}
public static void allBeh (LeftAngleTri lt1)
{
    lt1.rotate();
    lt1.getArea();
    lt1.fillColor();
}
public static void allBeh (EquilateralTri et1)
{
    et1.rotate();
    et1.getArea();
    et1.fillColor();
}
}
```

* A Method which can handle single type of object is called Specialized method and the process is called "specialization."
* Specialization avoids repeatative codes to certain extent.
* problem with Specialization is "everytime newtype of object is added to the application design". We have to develop corresponding Specialized method.
* for example : if Rhombus type of object is added. We have to create Specialited method for rhombus. this process is continous for every new type of shape.
* * To overcome above problems We can go for "Generalization."

Level-3 :-

```
class DrawingApp
{
  public static void allBeh(Circle c1) ⟶ Specialized Method
  {
    c1.rotate();
    c1.getArea();
    c1.fillColor();
    c1.getCircumference();
  }
  public static void allBeh(Rect r1) ⟶ Specialized Method
  {
    r1.rotatec);
    r1.getArea();
    r1.fillColor();
    r1.getperimeter();
  }
  public static void allBeh(Tri t1) ⟶ Generalized Method
  {
    t1.rotatec);
    t1.getArea();
    t1.fillColor();
  }
  public static void main(string[] args)
  {
    allBeh(new Circle(5));  ⟷ Circle c1 = new Circle(5);
    allBeh(new Rect(6,5));  ⟷ Rect r1 = new Rect(6,5);
    allBeh(new RightAngleTri(1,3));  ⟵ Tri t1 = new RightAngleTri(1,3);
    allBeh(new leftAngleTri(2,5));  ⟵ Tri t1 = new leftAngleTri(2,5);
    allBeh(new EquilateralTri(1,4));  ⟵ Tri t1 = new EquilateralTri(1,4);
  }
}
```

↓ this type of object creation can be written as
A a1 = new A();
A a2 = new B();

* Right Angle Triangle, Left Angle Triangle and Equilateral Triangle these methods can be creating Single Generalized method.

* Generalized method can handle any type of triangle.

** Runtime polymorphism :-

Combination of Generalization, upcasting and Overriding.

11/04

Create a final Generalized Method for Every type of Shape in the mini project.

Level : 4 :-

```
class DrawingApp
{
 public static void allBeh (Shape S1)
 {
    S1.rotate();
    S1.getArea();
    S1.getfillcolor();
 }
   if (S1 instance of Circle)
   {
     Circle C1 = (Circle)S1;
     C1.getCircumference();
   }
  else if (S1 instance of Rect)
  {
     Rect r1 = (Rect)S1;
     r1.getperimeter();
  }
}
```

```
Public static void main (String[] args)
{
    allBeh ( new Circle(5));
    allBeh (new Circle(7));
    allBeh (new Rect(1,3));
    allBeh (new Rect (2,5));
    allBeh (new RightAngleTri (5,7));
    allBeh (new RightAngleTri (7,9));
    allBeh (new LeftAngle Tri (9,11));
    allBeh (new LeftAngle Tri (7,8));
    allBeh (new EquilateralTri(1,2));
    allBeh (new Equilateral Tri (5,4));
  }
}
```

* By combining Generalization, method overriding and upcasting we can achieve runtime polymorphism.
* Runtime polymorphism says call to overrided method is resolved during runtime based on the type of object created.
* for example: Generalized method all behaviour, decides "which overrided getArea() and fillColor() to call based on the type of object passed to Generalized method."

→ Points to remember in case of Generalization

(i) Generalization method can handle multiple types of objects
(ii) Generalized method argument type will be Super class or interface reference variable.
(iii) In case of generalized method every object that is passed will be ~~down~~ casting. upcasted.

(iv) within generalized method we can easily refer super class properties.

(v) within generalized method, we need to downcast the given object inorder to refer subclass properties.

(vi) Before downcasting we need to validate the type of object by using instanceof keyword.

Syntax:

Refvar instanceof className;

Compile time polymorphism :—

call to overloaded methods is resolved during compile time based on the type of argument passed is called compile-time polymorphism.

Inorder to achieve compile time polymorphism method overloading is compulsary.

Refer Method overloading programming example.

i.e. an example for compile time polymorphism.

Date : 29/03/19

Interface :—

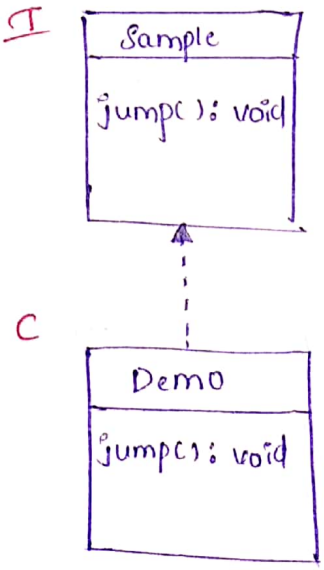
* It is one of the Type definition blocks in java.
* Interface naming format is similar to class naming format.
* Interface doesnot allows complete methods rather it can allows only abstract methods.
* Interface methods are bydefault abstract. hence we need not declare method as abstract

  abstract public void move(); / public void move();
* Interface methods must be implemented within the class. hence class is called as "Implementation class."
* *** [Inorder to use methods of interface, we need an object of implementation class.] ***
* class implements interface.
* Interface objects cannot be created.

Program :—

```
interface Sample
{
  public void jump();
}
class Demo implements Sample
{
   public void jump()
   {
     S.O.pln("jump into water...");
   }
}
class Interfaceprg1
{
  public static void main(String[] args)
   {
     Demo d1 = new Demo();
     d1.jump();
   }
}
```

UML diagram :—



new operator navigates class to interface loads signature then comes back to class loads implementation.

* Within interface all the variables are final and static

prog:- interface Sample1

```
{
  [int a = 60;] ⟶ final & static
}

class Interfaceprg2
{
  public static void main(String[] args)
  {
    S.O.pln("main starts....");
    S.O.pln(Sample.a);
    S.O.pln("main ends...");
  }
}
```

output:-

```
main starts
60
main ends.
```

* Constructors are not allowed within interface.

01/04/19:-

* Different combinations of classes and interfaces.

Refer assignment book. (29/03/19)

(i) interface can extend another interface.

(ii) An interface can have multiple implementation classes. (&)
An interface can be implemented by multiple classes.

(iii) A class can implement multiple interfaces

(iv) An interface can extend multiple interfaces.

(v) A class can extend as well and class as well as implement interface.

01/04/19

(vi) If multiple interfaces have same abstract method we have to implement the method only once.

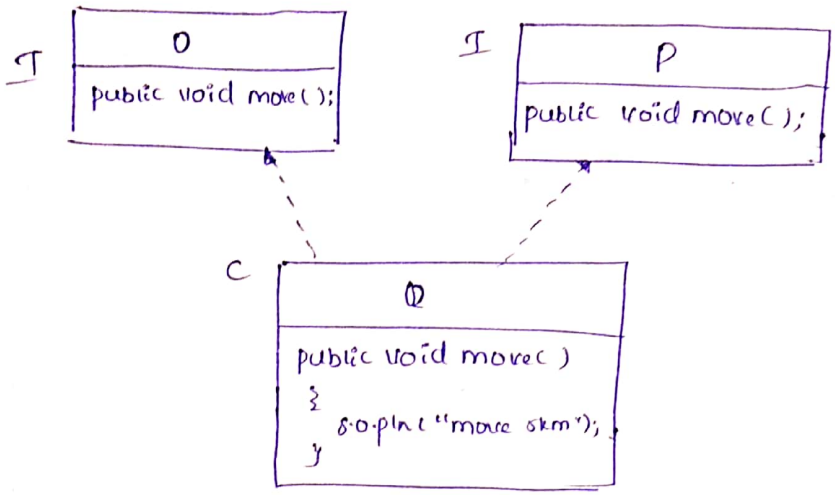
02/04/19

Multiple Inheritance :- (in interface)

* Multiple inheritance is possible through interface because

(i) There is no constructor chaining problem because there is no constructor within interface

(ii) There is no diamond problem because interface doesnot allows implementation. incase multiple interface have same method, it has only method signature. and these methods will be implemented only once.



Derive Datatypes :- (or) Non-primitive Datatypes (or) user defined Datatypes :-

* Primitive datatypes are restricted to single type of data. for example
  int represents integer type of data.
  char represents single character type of data
* To represent multiple type of data using single datatype, we have to create our own datatype. this is called user defined datatype.
* Derived datatypes can be created by using class and interface. hence class and interface are called type-definition blocks.
* Derived datatype name is similar to class name (or) and interface name.

* using derived datatype we can create derived variables. (reference var).
* using derived variable we can store objects address.

Type Definition blocks	Derived Datatype	Derived DT
		derived Ref var (ref var);

class Rect
{
=====
};
	Rect	Rect r1;

class Circle
{
=====
};
	circle	Circle c1;

class interface Animal
{
=====
};
	Animal	Animal a1;

interface car
{
=====
};
	car	car c1;

04/04

Lifecycle of Object :-

* Objects are created by using new operator
* Objects are used by the programmer.
* If object is no longer useful, it is better to make object as abandon. abandon
* Object whose control is no longer with the programmer (or) Object which is doesnot have any reference variable. Those objects are called as abandon objects
* By making object as abandon, it becomes eligible for Garbage Collection

* There are two ways to make abandon

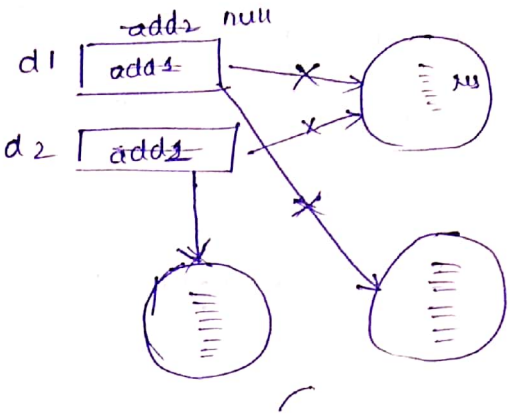
(i) By reinitializing the reference variable with another object old object becomes abandon

(ii) By reinitiacize the reference variable with null value. object becomes abandon.

* When Garbage collector removes abandon object program performance is improved.

```
class Dog
{
  ------
  ------
  ------
  ------
}
```

```
Dog d1;
Dog d2;
d1 = new Dog();
d2 = d1;
d1 = new Dog();
d2 = new Dog();
d1 = null;
```



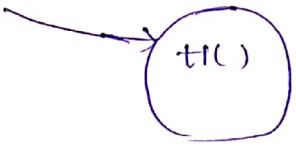
→ There are two abandon objects.

Type Casting :-

object type casting :- [Non-primitive type casting / Derived type casting]

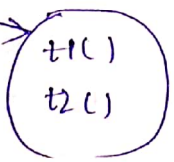
Converting one object type to another object type is called object type casting.

```
A a1 = new A();
```



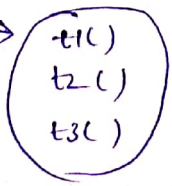
Atype

```
B b1 = new B();
```

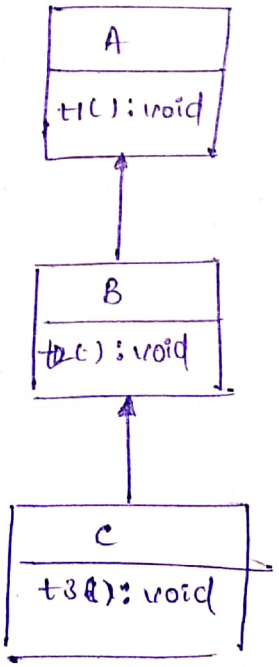


B type

```
C c1 = new C();
```



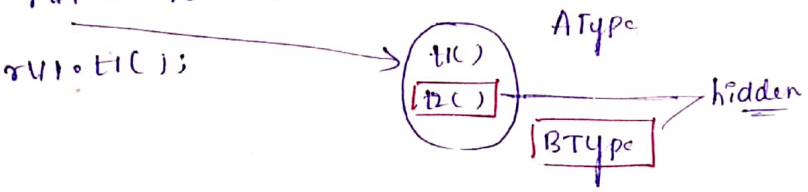
C type



Upcasting :-

* Converting subclass object type to superclass object type is called upcasting.
* When object is upcasted, subclass properties are hidden, super class properties are shown.
* upcasting is possible because subclass properties & object will have properties of superclass.

```
A rv1 = new A() new B();
rv1.t1();
```



Down casting :-

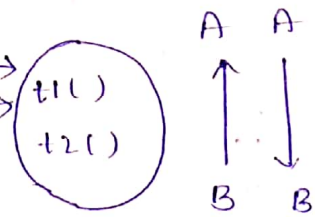
* Converting Superclass object type to sub class object type is called Down casting.

```
[B rv1 = (B) new A();]
```

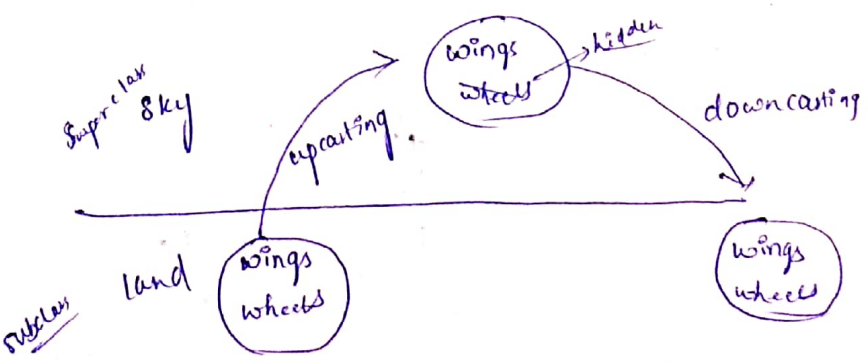
* This type of down casting is not possible because superclass object will not have properties of sub class.
* Down casting is possible only if object is upcasted.
* During down casting hidden subclass properties are becomes visible

Ex:

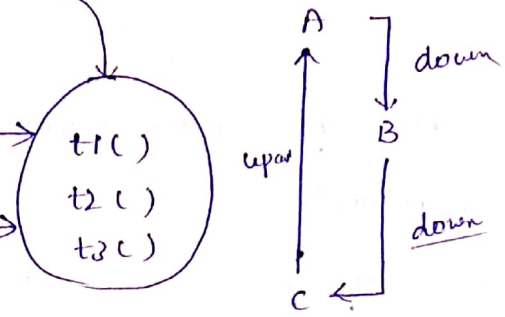
```
A rv1 = (A) new B();
rv1.t1();
B rv2 = (B) rv1;
rv2.t1();
rv2.t2();
```



Ex: Aeroplane



```
A  rv1 = (A) new C();
   rv1.t1();

B  rv2 = (B) rv1;
   rv2.t1();
   rv2.t2();

C  rv3 = (C) rv2;
   rv3.t1();
   rv3.t2();
   rv3.t3();
```



→ Single object three different perspective.

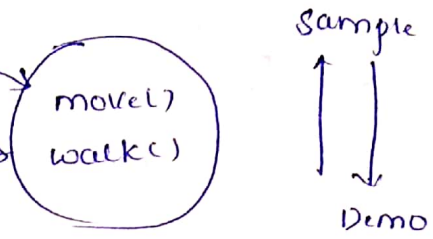
* ~~from~~ rv1 perspective object looks like A Type
* from rv2 perspective object looks like B Type
* from rv3 perspective object looks like C Type.

Date : 08/04/19

Program :-

```
class sample
{
   public void move()
   {
      S.O.pln("move 5km");
   }
}
class Demo extends Sample
{
   public void walk()
   {
      S.O.pln("walk 3 km");
   }
}
```

```
class objectcasting prg1
{
    public static void main(String[] args)
    {
implicit ← Sample rv1 = new Demo();  //(or) Sample rv1 = (Sample) new Demo();
                                          ↑ not required because implicit
         rv1.move();
explicit ← Demo rv2 = (Demo) rv1;
         rv2.move();
         rv2.walk();
    }
}
```



* upcasting is implicit in nature because at any point of time every single subclass will have only one super class. hence programmer need not explicitly specify "which superclass type to convert"

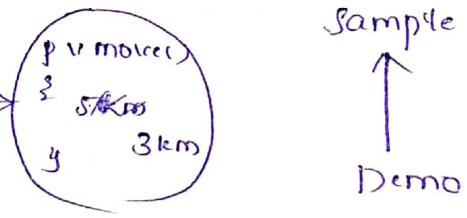
* downcasting is explicit in nature because single superclass can have multiple subclasses. hence programmer has to explicitly specify "which subclass type to convert."

Realtime example :- Relati-father & childrens.

∴ We are 3 childrens if someone is asked what about your parents / father there is no confusion to give answer because only one parent is there. [Implicit] Suppose if anyone is asked to my parent what about your children, my parent gets confusion and ask to that person which children (?) [Explicit]

Program 2 :-

```
class Sample
{
    public void move()
    {
        S.O.pln("move 5km");
    }
}
class Demo extends Sample
{
    public void move()
    {
        S.O.pln("move 3km");
    }
}
class Object Casting prg2
{
    public static void main(String[] args)
    {
        Sample s1 = new Demo();
        s1.move();
    }
}
```



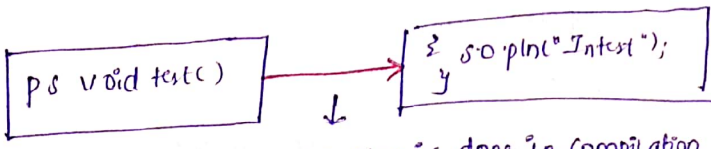
* When object is upcasted overrided methods will not be hidden. because method belongs to superclass. but if method is called from superclass reference variable, subclass implementation gets execute because method is overrided.

Method Binding :-

* Connecting Method signature with Method implementation is called Method Binding.

→ Method binding is not done in coding part

Progs:
```
class A
{
    public static void test() → Method signature
    {
        s.o.pln("In test"); } → Method implementation
    }
}
```



This binding is done in Compilation time.

Compile time Binding :-

When the methods are binded in during compilation time

(i) Compile Time Binding.
(ii) Static Binding.
(iii) Early Binding.

Static Binding :-

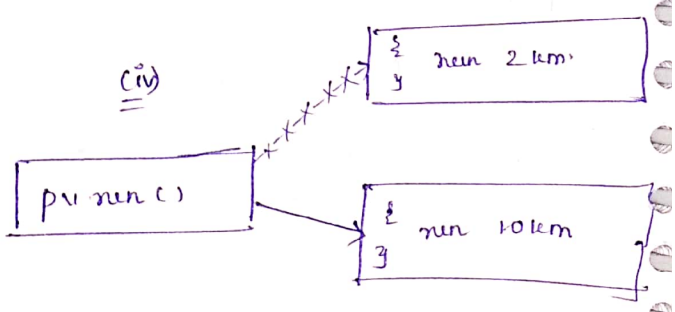
When the methods are binded only one time (constant) cannot be changed

Early Binding :-

Binding is done before execution process.

Progs:
```
class B
{
    public void run()
    {
        s.o.pln("run 2kms");
    }
}
class C extends B
{
    public void run()
    {
        s.o.pln("run 10kms");
    }
}
```

(iv)



Run time binding
Dynamic binding
Late binding.

* private methods are binded during compile time.

Runtime binding :-

Binding is done during Runtime as JVM is done the binding.

Dynamic Binding:

It changes accordingly when ever the object is created.

Late Binding :-

During the execution process if the object creation line is there in the nth line then the binding takes time as JVM is the interpreter goes one line after the other.

(iv) → Here the binding is done, when the object is created. if the Superclass object is created then the binding is done with the Super class implementation and vice versa after the super class binding. the subclass binding will be done after removing the connection of super class binding.

09/04/19

* Static and final methods are executed in compilation time and non static methods are executed during runtime.

Primitive Type Casting :-

* Converting one primitive data type into another primitive data type is called primitive type casting.
* There are two types.
  (i) Narrowing (ii) Widening.

(i) Narrowing :-

* Converting Bigger primitive data type into smaller primitive data type is called Narrowing.
* In case of narrowing there is always a loss of data. Hence, it is unsafe casting. Sin.
* Since there is a loss of data explicit permision must be given by the program.

Ex:- `int i = (int) 7.4;` 7.4 → double

i | 7 |

`S.O.pln(i);` ⟶ o/p: 7

(ii) **Widening :-**

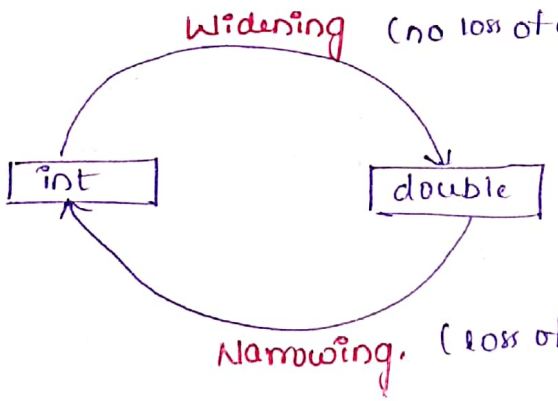
* Converting smaller primitive datatype to bigger primitive data type is called Widening.
* In case of widening there is no loss of data, hence it is safe casting.
* Since there is no loss of data explicit permission is not required by the programmer rather it is implicit in nature.

Ex: `double j = 9;` 9 → integer value

j | 9.0 |

`S.O.pln(j);` ⟶ o/p: 9.0

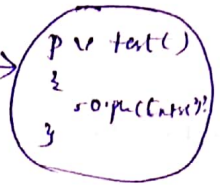
Summary:-



Program :-

```
interface A                    // Interface class
{
   public void test();
}
class B implements A           // Implementation class.
{
   public void test()
   {
      S.O.pln("In test...");
   }
}
```

```
class Objectcastingprg3
{
    public static void main(String[] args)
    {
        A rvl = new B();
        rvl.test();
    }
}
```



* upcasting is not only done in subclass to superclass also but also done in implementation class to interface class.
* Interface class cannot creating a object because interface can be declared as abstract but we can create reference variable for interface

```
A rvl = new B();
```

A → interface class
B → implementation class.
rvl → reference variable

* Reference variable (rvl) holds the address of interface (A class)
* We can create reference variable for interface.
* using interface reference variable we can hold implementation class object.

Note:- using Interface reference variable when interface methods are called, implementation present in implementation class gets executed.

* If there is an upcasting between implementation class and interface, methods of interface will not be hidden (Boverrided methods are not hidden)

[Write a program to create two interfaces and one class.
(i) class implements interface, interface extends interface.] Wrong

call to overloaded

13/04/19 :-

Access Specifiers :-

Using Access specifiers we can control access level & visibility level of,

(i) class
(ii) interface
(iii) static variable
(iv) Non static variable
(v) static Method
(vi) Non static Method
(vii) Constructor.

There are four types of Access Specifiers.

1. public
2. private
3. protected
4. default (no keyword)

1. public :-

public members can be used,

(i) within the same class
(ii) within the other classes but those classes should belong to the same package.
(iii) within the other classes of different package but we must include import statement.

→ import packagename . classname ⟶ Syntax.

* import statement should be added after package declaration before class declaration.

Program :-

(i) Package demo1;

```
public class A
{
  public static int x = 10;
  public int y = 20;
  public static void main (String[] args)
  {
    S.O.pln (x);
    A rv1 = new A();
    S.O.pln (rv1.y);
  }
}
```

(ii) Package demo1

```
public class B
{
  PSVM (S[] args)
  {
    S.O.pln(A.x);
    A rv1 = new A();
    S.O.pln (rv1.y);
  }
}
```

(iii) Package demo2

```
import demo1.A
public class C
{
  public static void main (String[] args)
  {
    S.O.pln (A.x);
    A rv1 = new A();
    S.O.pln (rv1.y);
  }
}
```

2. Default :-

default members can be used,

(i) within the same class.

(ii) within the other class but class should belong to the same package.

(iii) if class is default it cannot be imported.

(iv) Default is also called as package level access specifier.

3. private :-

* private members can be used only within the same class.
** class cannot be declared as private.
* private members cannot be inherited.
* private methods cannot be overrided

4. protected :-

protected members can be used,

(i) within the same class.
(ii) within the other class but of same package.
(iii) within another class of another package after inheritance. and protected members should be access through subclass.

* class cannot be declared as protected.

(i)
```
package demo1
public class A
{
  protected int y = 20;
  psvM (S[] args)
  {
    A rvl = new A();
    S.O.pln (rvl.y);
  }
}
```

(ii)
```
package demo1
public class B
{
  psvm (S[] args)
  {
    A rvl = new A();
    S.O.pln (rvl.y);
  }
}
```

(iii)
```
package demo2
import demo1.A
public class c extends A
{
  psvm (S[] args)
  {
    C rvl = new C();
    S.O.pln (rvl.y);
  }
}
```

(i) Package sample1

```
{
    public class M
    {
        public static int x = 45;
        public int y = 65;
    }
}
```

(ii) package sample2

```
{
    public class M
    {
        public static int x = 96;
        public int y = 70;
    }
}
```

(iii) Package sample3

```
{
    import sample1.M;
    // import sample2.M; // Error.
    public class M
    {
        public static void main (String [] args)
        {
            S.o.pln (M.x);
            M rv1 = new M();
            So.pln(rv1.y);
            S.o.pln (sample2.M.x);
            Sample2.M rv2 = new sample2.M(); // fully qualified class-
            S.o.pln(rv2.y);
        }
}
```

* When we use multiple classes of different class names we can use multiple import statements.
* When we use multiple classes of same class name we can use only one class through import statement, all other classes must be used by using fully qualified class name. (package name.classname).
* To access static member
  → packagename.classname.static member name.
  Ex: Sample2.M.x
* To create object and access nonstatic member
  → packagename.class name refvar = new packagename.classname (); (↑ Constructor)
  Ex: Sample2.M rv1 = new Sample2.M();

Note 1 :- Single java file.

```
class A
{
}
class B
{
}
class C
{
}
```

A.java (✓)
B.java (✓)
C.java (✓)
X.java.

* If java file consists of default access specifier classes then javafile name can be anything.

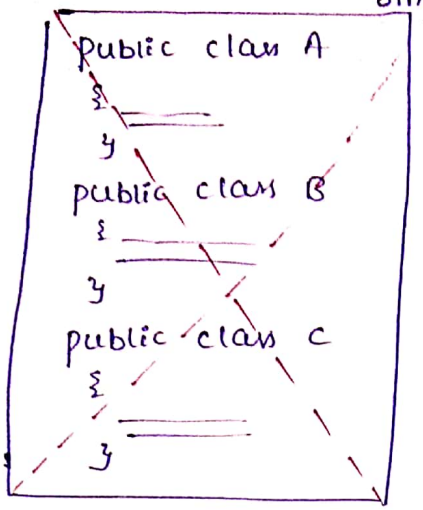
Note 2 : Single java file

```
class A
{
}
public class B
{
}
class C
{
}
```

A.java (✗) (✓)
B.java (✓) (✓)
C.java (✗) (✓)
X.java (✗)

* If java file consists of a public class then it is compulsory java file name must be similar to public class name

* Note 3:-



Not possible, because java file can have only one public class. Multiple public classes in a single java file is cannot possible.

16/04/19:

Encapsulation :-

Protecting datamembers and giving access to datamembers through member functions is called encapsulation.

(or)

Binding Data members and Member function into a single unit class is called encapsulation.

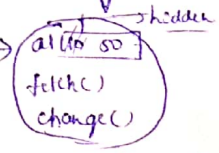
(or)

Combining Datamembers and Member function into a single unit class is called encapsulation.

Program :-

```
public class A
{
    private int i = 10;
    public int fetch()
    {
        return this.i;
    }
    public void change()
    {
        this.i = i;
    }
}
```

```
public class Test
{
    public static void main(String[] args)
    {
        A a1 = new A();
        S.o.pln(a1.i); // not possible(Error)
        S.o.pln(a1.fetch());
        a1.change(50);
        S.o.pln(a1.fetch());
    }
}
```



output: 10
50

```
public class cal
{
   private int monthNum;
   public int fetch()
   {
      return this.monthNum;
   }
   public void change(int monthNum)
   {
      if(monthNum >=1 && monthNum <=12)
      {
         this.monthNum = monthNum;
      }
      else
      {
         throw Exception;
      }
   }
}
```

* We go for encapsulation to protect our datamembers from invalid values.

→ for example : if monthNum is not protected (encapsulated), user may misuse the datamember by assigning invalid values. invalid values for MonthNum are not between 1 & 12.

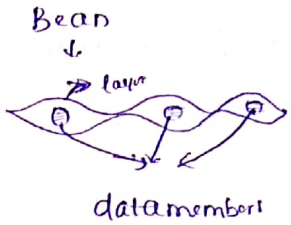
* If datamember is not protected (data member is public), It is a under user control i.e. user will decide which value to assign but that value can be a valid value & a Invalid value.

* If datamember is protected (data member is private), it is under developers control that is developer decide which value to allowed.

Java Bean class :-

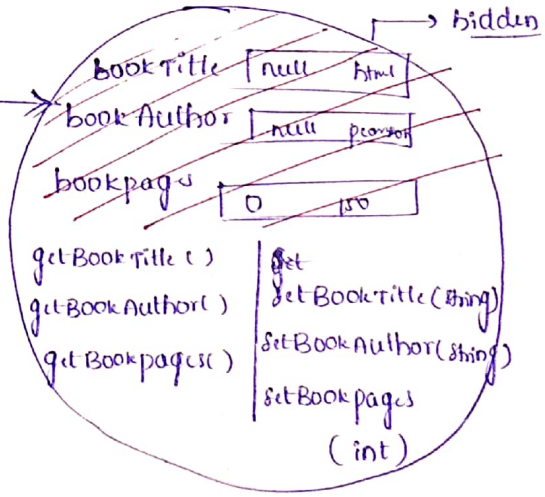
```
public class Book
{
    private String bookTitle;
    private String bookAuthor;
    private int bookpages;

    public String getBookTitle()
    {
        return this.bookTitle;
    }
    public String getBookAuthor()
    {
        return this.bookAuthor;
    }
    public int getBookpages()
    {
        return this.bookpages;
    }
```

```
public void setBookTitle (String bookTitle)
 {
    this.bookTitle = bookTitle;
 }
public void setBookAuthor (String bookAuthor)
 {
    this.bookAuthor = bookAuthor;
 }
public void setBookpages (int bookpages)
 {
    this.bookpages = bookpages;
 }
}
```



user's logic

```
class Bookstore
{
  public static void main (String[] args)
  {
    Book bl = new Book();
    bl.setBookTitle ("html");
    bl.setBookAuthor ("pearson");
    bl.setBookpages (150);
    S.O.pln (bl.getBookTitle());
    so.pln (bl.getBookAuthor());
    S.O.pln (bl.getBookpages());
  }
}
```



* Developing a class where each datamember is declared as private and for each datamember there is a seperate getter and setter method. These type of classes are called Java Bean classes.

Date :- 16/04/19

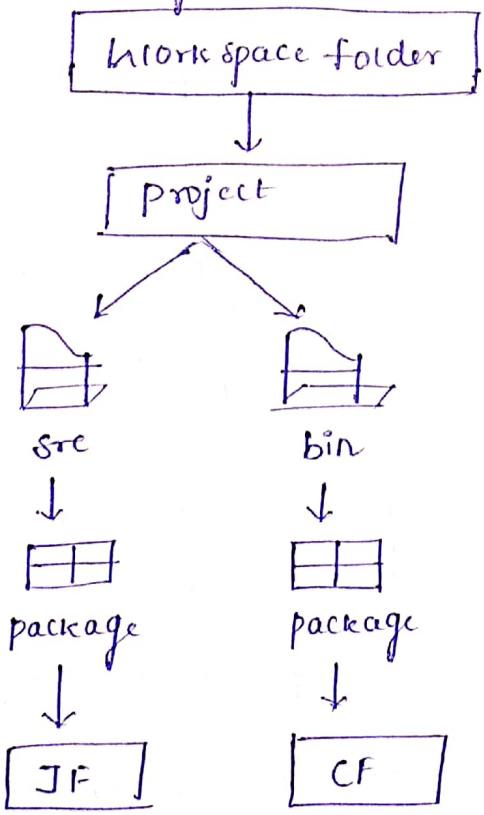
→ We go for JavaBean class to control the our Datamember that is Developer will have flexibility to choose which data member requi should have only read access (student id, empid, aadhaar no etc...), which data member should have only write access (atm pin no, online account passwords etc...] , which data members should have both read and write access (student marks, employee salary, mobile number etc..] .

* JavaBean clases are use for developing business application [EJB application — Enterprise JavaBean application] · and Java Server pages (JSP's] .

Working with EclipseIDE :-

Integrated Development Environment.

Eclipse hierarchy :-



open source

free

* Single workspace can have multiple projects.
* project folder can Cont
* under source folder multiple packages we can be created.
* Same packages are replicated under bin folder by eclipse.
* Single package can have multiple classes and interfaces interfaces

Configuring Eclipse :-

Step 1 :- My computer → any drive → create a new folder → [[java project work space]

Step 2 :- open eclipse folder → double click on eclipse.exe file → use browse button and select java project work space folder. → launch. (click on) → close welcome tab.

Step 3 :- click on file → new → project → double click on java → jdouble click on java project → provide project name. (projectTECH4) → click on finish → click on open perspective.

Note :- as soon as project is created source (src) folder and bin folder are created automatically by eclipse.

Step 4 :- double click on created project → Right click on src folder → new → package → provide package name → click on finish

Step 5 :- Right click on created package → new → class (or) interface → provide the class name (or) interface name

→ In eclipse Compilation is automatic and all the class files are Stored under the package which is present in bin folder.

Step-6: By clicking run button (▷) program will execute.

K. Sandhya Sri

K. Sandhya Sri

K. Sandhya Sri