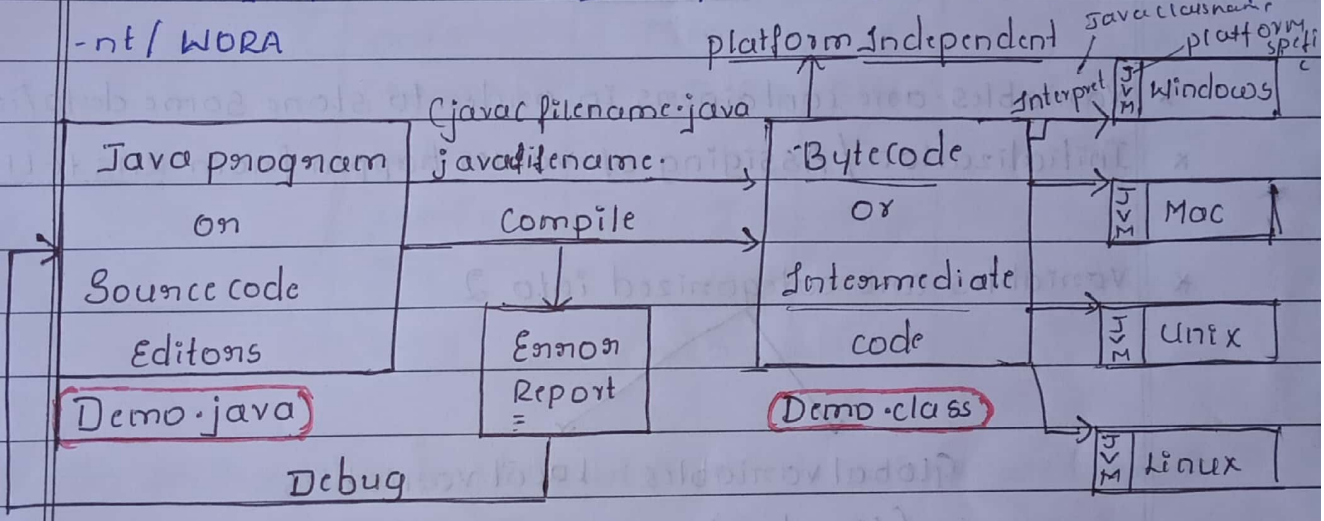
Date: 21/01/20

Java

* Java is a high level programming language
* programming language is a medium to interact with s/m
* Highlevel language is a language in a normal English i.e Human understandable form
* James Gosling was a person who introduced Java
* The company which started java is SunMicrosIm (system)
* Currently Java is owned by oracle.

Working of a Java Program / How is java platform independent / WORA



JVM : Java virtual Machine (platform dependent)

WORA : Write Once Run Anywhere

Bytecode is an intermediate which is neither low-level nor highlevel language so it uses jvm → ① convert to machine level ② Execute line by line

* Firstly we build the java program using editors and save it with the extension .java

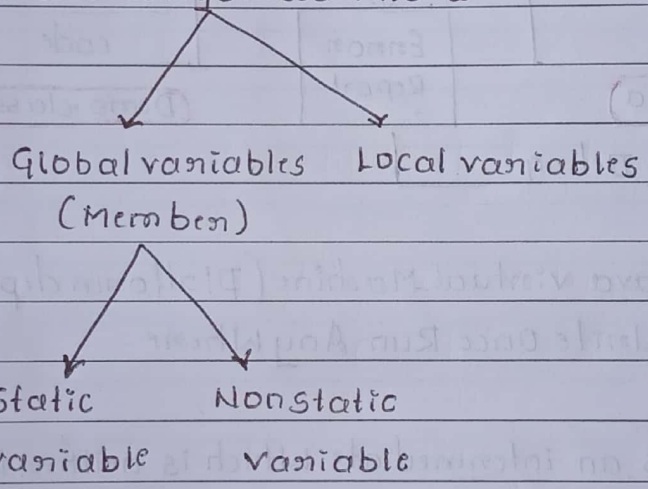
* once we are done developing the program we need to compile it

Date: / /

* compilation is a process in order to check if there are any errors in my java program or not
* If compilation is unsuccessful we get error report based on error report we need to debug the program
* If compilation is successful we generate bytecode which is intermediate code / platform independent code
* Extension of all bytecode is .class
* This bytecode can be executed on all platform i.e all operating sy

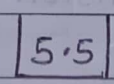
Variables

* Variables are containers in order to store some data/inform
* Initialization / Assiging of values happen from RHS to LHS
* Variables are categorized into 2



Example : Age = 25
LHS RHS

Height = 5.5



V => LHS RHS

Datatypes

* Data types are used to indicate or specify the type of data stored into a variables
* Datatypes are categorized into 2
  1) primitive Datatype
  2) Non primitive Datatype
* In order to store non decimal numeric values we have the following datatypes
* The difference between those datatypes is Memory size

Data types	Memory Size	
	Bytes	Bits
byte	1	8
short	2	16
int	4	32
long	8	64

* In order to store decimal numeric values, we have the following datatypes

Data T	Memory size	
float	4	32
double	8	64

widely used and default datatype for decimal.

* In order to store true / false we have following datatype

Data T	Memory size	
	Bytes	Bits
Boolean	1	8

* In order to store a single character, we make use of char
* char data should be enclosed within 'single quotes' (' ')
* The Memory size of char is 2 bytes = 16 bits

Note : All the above mentioned 8 data types are together called as primitive data types.

String : It is a dataStri type to store a sequence of characters

* String data has to be enclosed within "double quotes"

Note : Java is CaseSensitive Where in lowercase letters are not equivalent to upper case letters/values (a ≠ A)

22/01/2020

Variable Declaration

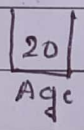
Syntax:

```
datatype VariableName;
int age;
double salary;
```

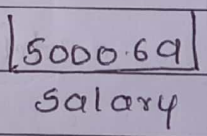
Variable Initialisation

Syntax :

```
VariableName = value;
age = 20;
```

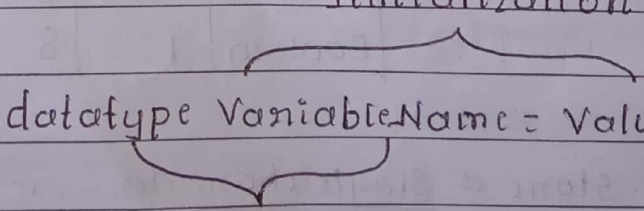



```
Salary = 5000.69
```



Variable Declaration & Initialization

Syntax :




```
boolean x = true;
          false;
```

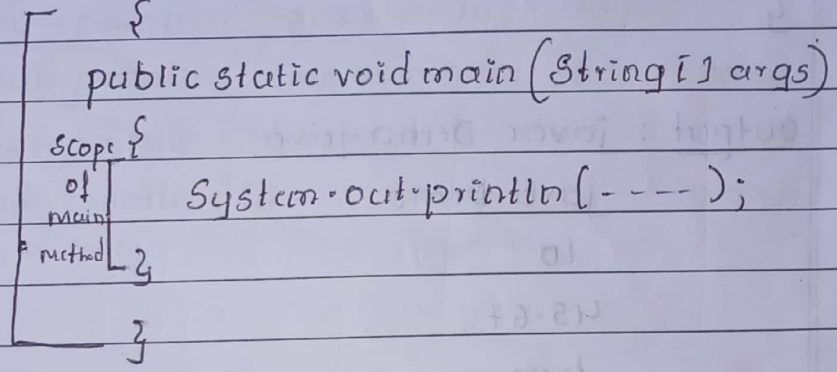
Date: / /

String subject = "java";
= "ljsp2020", (string can store number, character but it should be within " " quoted)
char gender = 'M';

Structure of a java program:

1) Class
2) Main method
3) Print Statement

(i) class: Class className



The filename should be same as classname during file saving className.java

Ex:1) class Firstprogram
{
Public static void main (String[] args)
{
System.out.println ("Hello world");
}
}

Output: Open cmd
cd desktop
cd java programs
javac Firstprogram.java - (compiln)
java Firstprogram - (Interprtn)
Hello world!!!

// cd:- change directory to enter to the save files

Ex:2)
```
Class Demo
{
Public static void main (String[] args)
{
System.out.println (10);
System.out.println (45.67);
System.out.println (true);
System.out.println ('2');
System.out.println ("Ijsp@2020");
}
}
```

Output : javac Demo.java

```
java Demo
10
45.67
true
2
Ijsp@2020
```

Ex3) Write a java progrnam to follow the below statement / Scenarios

i) Create class called as student

ii) Define main method

iii) Under main method initialize 2 variables called as name and age entering those respective values

→
```
Class student
{
public static void main (String [] args)
{
String name= "Bhagya";
int age = 23;
System.out.println ("name");
system out. println (age);
}
}
```

O/P

Bhagya

23

Date: / /

Note: In java, in order to perform Concatenation we make use of '+' operator

Q)
```
class Employee
{
public static void main (String[] args)
{
 int id = 101;
 String name = "Jerry";
 double salary = 123.45;
System.out.println("Employee Id:" +id);
System.out.println("Employee name is:" +name);
System.out.println("Employee Salary =" +salary);
System.out.println(id +" " +name+ " " +salary);
}
```

Output

```
javac Employee.Java
java Employee
Employee Id=101
Employee Name is : Jerry
Employee Salary = 123.45
101 Jerry 123.45
```

23/1/2020

Operators

1) Arithmetic Operators
2) Assignment Operators
3) Relational / Conditional / comparsion Operators
4) Logical Operators
5) Unary Operators

1) Arithmetic Operators

+ : Addition
- : Substraction
* : Multiplication
/ : Division
% : Modulus

5 → divisior

2 | 10
  | 10
  | 0 → Modulus

Ex: 10/2 = 5 10%2 = 0

Example:

```
Class AnithmeticOperators
{
public static void main (String[] args)
  {
    int x=10;
    int y=20;
    int sum = x+y;
    int diff = x-y;
System.out.println("Sum = "+sum);
System.out.println("Difference is " + diff);
System.out.println (y*5);
System.out.println (30/3);
System.out.println (30%3);
```

Output:-

```
Sum = 30
Difference is = -60
100
10
0
```

Date: / /

2) Assignment Operators

=

+=

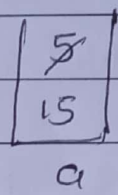
-=

*=

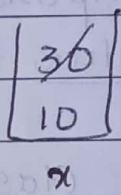
/=

%=

```
int a=5;
  a= a+10  or  a+=10
  a=15
```




```
int x=30
  x-=20
  x=x-20
  x=30-20
  x=10
```



Ex: Class Assignment Operator

```
{
public static void main (String[] args)
{
 int x=10;
 System.out.println("value of x is :" +x);
    x+=20;
 System.out.println("value of x is :" +x);
 System.out.println("=====");
 int a= 6
 system.out.println("value of a is:" +a);
    a*=5;
 system.out.println("value of a is:" +a);
}
}
```

O/P

Pathname → cmd
= Assigning
== comparsion

papergrid

Date: / /

Output

Value of x is 10
value of x is 30
= = = = = =
valur of a is 6
value of a is 30

3) Relational / conditional / comparsion Operators

< : lesser than
> : Greater than
<= : lesser than or equal to
>= : Greater than or equal to
== : Equals to
!= : not equal to

Note : Comparsion Operators will always return boolean values i.e (true/false)

Ex:

```
Class comparsion operator
{
public static void main (String[] args)
    {
        int x=10;
        int y=20;

        boolean result1 = x<y;
        boolean result2 = y>x;
System.out.println(result1 + "\t" + result2);
System.out.println(" = = = = ");
System.out.println ( x<=10);
System.out.println ( 3>=4);
System.out.println (" - - - - ");
System.out.println ( x==10);
System.out.println (y==30);
System.out.println ("- - - - ");
System.out.println (x!=10);
System.out.println (y!=30);
```

O/P

true true
---
true
false
---
true
false
---
false
true

4) Logical operators

AND => &&
OR => ||
NOT => !

return => boolean { true = 1, false = 0 }

AND		&&
T	T	T
T	F	F
F	T	F
F	F	F

OR		||
T	T	T
T	F	T
F	T	T
F	F	F

NOT	!
T	F
F	T

Ex:- Class logical operators

```
{
public static void main (String[] args)
{
 int x=10;
 int y=20;
 boolean result1 = x<y && y>x;
 boolean result2 = x<y && x==1;
System.out.println (result1);
System.out.println (result2);
System.out.println ("---");
System.out.println (1<2||2>10);
System.out.println (2<1||2==3);
System.out.println ("---");
System.out.println (!true);
System.out.println (!false);
System.out.println ("---");
System.out.println (!(1<2));
}
}
```

O/P

```
true
false
---
true
false
---
false
true
---
false
```

5) Unary Operators

++ (increment by 1)

-- (Decrement by 1)

++ → pre increment
   → post increment

-- → pre decrement
   → post decrement

Ex:
```
Class Unary
{
public static void main (String[] args)
  {
    int x = 5;
    System.out.println ("x:" +x);
     x++;
    system.out.println ("x:"+x);
      ++x;
    System.out.println ("x:"+x);
      x--;
    system.out.println ("x:"+x);
      --x;
    System.out.println ("x:"+x);
      }
}
```

Output ⇒
```
x:5
x:6
x:7
x:6
x:5
```

(1)
```
int x= 10;
int y= x++;
```
post increment → First assign, then increment

x [10/11] y [10]

(2)
```
int a= 5
int b= ++a;
```
pre-increment → First increment, then assign

a [5/6] b [6]

Date: / /

(3) int i=2
j=i--;
post decrement
First assign, then decrement
i [1] j [2]

(4) int p=50;
int q=--p;
pre-decrement
First decrement, then assign
p [50 49] q [49]

(a) int a=5
int b=++a;
pre increment
First increment, then assign

Ex) class Unary
{
psvm (String[] args)
{
int q=123;
int w=q++;
System.out.println(q+" "+w);

System.out.println("-----");

int o=444;
int p=--o;
System.out.println(o+" "+p);

}
}

Decision / conditional statements

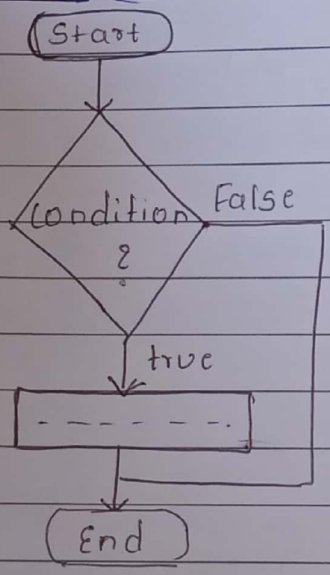
These statements are used to take some decision based on the condition specify. The different decision statements are as follows

1) Simple if
2) if else
3) if else if
4) Nested if
5) Switch

(1) Simple if

Simple if is a decision making statements where in we execute a set of instructions only if the condition is true

Ex: Flow chart



Syntax:

```
if (condition)
{
  statement 1;
  - - - - - -
  statement n;
}
```

(statement 1; ... statement n;) — Set of instructions

Ex:-

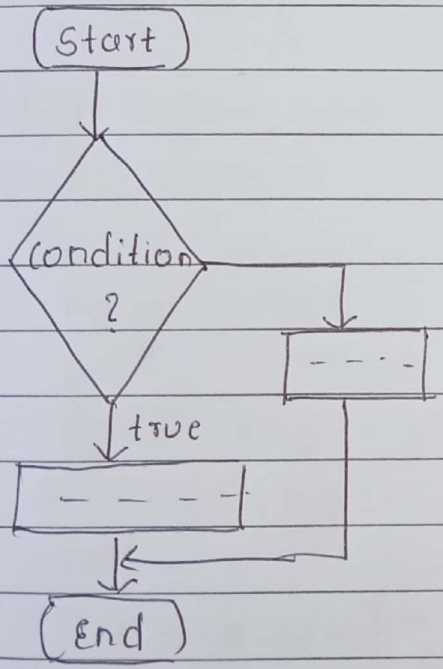
```
class SimpleifDemo
{
psvm (String[] args)
{
  System.out.println ("START");
}
    int a=10;
    int b=10;
```

```
if (a<=b)
{
system.out.println(a+" is less than or equal to"+b);
}
system.out.println("End");
}
}
```

(2) if else :-

x if else is a decision making statement where in, if the condition is true we execute if block. Otherwise if the condition is false we execute else block.

flow chart



Syntax :

```
if (condition)
{
statement 1;
- - - -
statement n;
}
else
{
- - - -
}
```

set of instructions

Ex

```
class IfelseDemo
{
  psvm(string[] args)
{
   s.o.pln("START");
   int x=100

   if (x<=10)
   {
s.o.pln(x+" is lesserthan or equal to 10");
   }

else
   {
   s.o.pln(x+" is gorcater than 10");
   }
    s.o.pln(" ------ ");

if(true)
{
 s.o.pln("HI");
 }
else
{
 s.o.pln("BYE");
}
 s.o.pln("----")

 if(false)
{
 s.o.pln("WELCOME");
 }
```

```
else
{
S.O.pln("Thank you");
}
S.O.pln("END");
}
}
```

o/p: START
100 is greater than 10
- - - - - - -
Hi
- - - -
Thank you
END

Q) Write a java program to find a number is +ve or -ve

→
```
class Number
{
psvm(String[] args)
{
    int x=5;
    if(x>0)
  {
    S.O.pln("x is a positive number");
  }
    else
    {
      S.O.pln("x is a negative number");
    }
}
```

2) Write a java program to find a number is even or odd

→
```
class Number1
{
psvm (String[] args)
 {
  int num=4;
  if (num % 2 ==0);
  {
   S.o.pln (num+" is a even number");
  }
  else
  {
   S.o.pln (num+" is a odd number");
  }
}
```

3) Write a java program to find maximum of 2 numbers

→
```
class Number2
{
 psvm (String[] args)
  {
   int x=5;
   int y=10;

   if (x>y)
{
 S.o.pln (x+" is a langer than"+y);
}

else
 {
 S.o.pln(x+" is a lesser than"+y);
 }
}
```

27/1/2020

3) if-else-if

if-else-if is a decision making statement where in we can check multiple conditions

Syntax:

```
if (condition)
{
  ---
  --
}
else if (condition)
{
  ----
}
else if (condition)
{
[
]
}
else
{
[
]
}
```

(else is optional)

Ex:-

```
class IfElseIfDemo
{
 psvm (String[] args)
 {
   if num = 250;
    if (num <=10)
      {
 S.O.pln (num + " is lessen than on equal to 10");
    }
```

```
else if (num<=20)
 {
 s.o.pln (num+" is lesser than or equal to 20");
 }
elseif (num<=30)
 {
 s.o.pln (num+" is lesser than or equal to 30");
 }
 else
 {
 s.o.pln ("Above conditions did not match");
  }
 }
}
```

O/P : Above condition did not match

0-34-F
35-59-Fc
60-100-Fc

Ex

```
class IfElseIfDemo1
{
 psvm (String[] args)
 {
  int marks=25;
  if(marks>=0 && marks<=34)
  {
    s.o.pln("fail");
  }
   else if (marks>=35 && marks<=59)
  {
    s.o.pln (" First class");
  }
```

```
else if (marks >=60 && marks <=100)
{
    s.o.pln("FCD");
}
else
{
    s.o.pln("Invalid Marks");
}
}
}
```

O/P : Fail

(4) **Nested if :**

Nested if is a decision making statement where in, one if is presented inside another if condition

Syntax:

```
if (condition)
{
  if (condition)
  {
    ---
    --
  }
  else
  {
  }
```

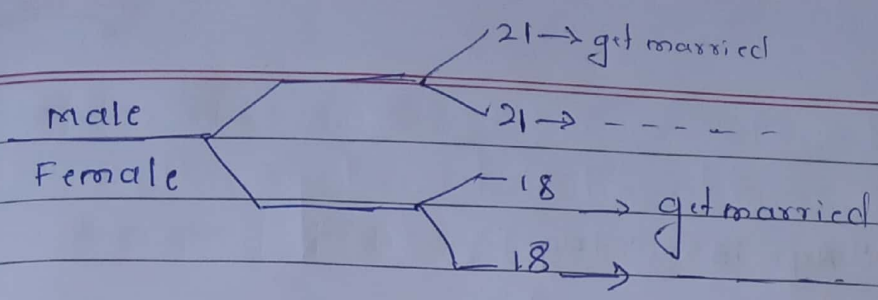
Ex ① class NestedIfDemo

```
{
psvm ( String[ ] args)
{
  char id = 'b';
  int password = 123;

  if (id == 'a');
  {
    S.O.pln (" user id is valid");
      if (password == 123)
      {
        S.O.pln (" password is valid ");
        S.O.pln (" login is Successful");
      }

  else
    {
      S.O.pln (" password is invalid");
              (" login is Unsuccessful");
    }
  }
  else
    {
      S.O.pln (" user id is Invalid");
              (" login is Unsuccessful");
    }
}
```

Output :- ① Userid is Invalid
login is Unsuccessful

② Userid is valid
Password is valid
login is Successful

Ex




```
class NestedIfDemo1
{
  psvm(String[] args)
  {
    char gender='M';
    int age = 24;

    if(gender=='M');
    {
      s.o.pln("Male");
      if(age>=21)
      {
        s.o.pln("Age is :"+age);
        s.o.pln("Get married & hopefully stay happy");
      }
      else
      {
        s.o.pln("age is :"+age);
        s.o.pln("Have patience");
      }
    else if(gender=='F')
    {
      s.o.pln("Female");
      if(age>=18)
      {
        s.o.pln("Age is :"+age);
      }
```

```
else
{
s.o.pln("Age is :" +age);
        (              )
}
}
else
{
 s.o.pln("Gender is Invalid");
}
}
```

(2) Gender is Invalid

o/p :- Gender is Male
age : 21
Get married & hopefully stay happy
stay happy

8) Write a java program to find largest of 3 numbers

```
class largestofthreeNumbers
{
 psvm(string[] args)
 {
    int a = 10;
    int b = 5;
    int c = 3;
 s.o.pln("a:"+a+" b: "+b+" c :"+c");
    if (a>b)
    {
     if (a>c)
     {
      s.o.pln("a is largest");
     }
```

```
else
{
  S.o.pln("C is largest");
}
}
else if (b>c)
  {
  S.o.pln("b is largest");
  }
else if (c>b)
  {
  S.o.pln("C is largest");
  }
  else
   {
    S.o.pln("Invalid");
    }
  }
}
  S.o.pln("----");

  if (a>b && a>c)
  {
    S.o.pln("a is largest");
  }
else if (b>a && b>c)
  {
  S.o.pln("b is largest");
  }
elseif (c>a && c>b)
   {
     S.o.pln("c is largest");
   }
```

a b c
a c
a or c

```
else
 {
 S.o.pln("Invalid")
   }
 }
}
```

Scanned with CamScanner

28/1/2020

(5) Switch statement :-
Switch is a conditional statement generally used for character comparsion

Syntax :- switch (choice/input)
{
case 1 : - - - - -
break;
case 2 : - - - - -
break;
- - -
:
case n : - - - -
default :

Ex :
```
class switchDemo
{
psvm (String[] args)
  {
    int choice=3;
      switch(choice)
  {
case 1 : s.o.pln("In case 1");
         break;
case 2 : s.o.pln("In case 2");
         break;
case 3 : s.o.pln("In case 3);
         break;
  default : s.o.pln("Invalid choice");
  }
}
}
```

O/p :- In case 3

Note: Break : is a keyword which is used to transfer the control outside the currently executing block

Ex:
```
class monthvalidation
{
psvm (String [] args)
{
s.o.pln ("start");
char month = 'Z';
switch (month)
{
case 'J' : s.o.pln ("In January");
           break;
case 'F' : s.o.pln ("In February");
           break;
case 'M' : s.o.pln ("In March");
           break;
default : s.o.pln ("Invalid month");
}
s.o.pln ("End");
}
}
```

Output: start
Invalid Month
End

Date: / /

Looping Statements

Looping statements are generally used to repeat a specific tasks

* looping statements are generally used to traverse the data as well
* The different looping statements are as follows
  1) for loop
  2) While loop
  3) do While loop
  4) nested for loop.

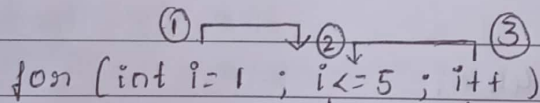
(i) for loop

for loop is a looping statement is used to repeat a task for the specified number of times

* We use for loop when we know the logical start and the logical end.

Syntax:
```
for (initialisation; condition; updation)
{
// set of instructions to be repeated;
}
```

Ex: print Hello world 5 times




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

Tracing:

i	i<=5	O/P
1	1<=5 → T	Hw
2	2<=5 → T	Hw
3	3<=5 → T	Hw
4	4<=5 → T	Hw
5	5<=5 → T	Hw
6	6<=5 → F	

Ex 6

Date: / /

```
Class ForloopDemo
{
public static void main (String[] args)
 {
  for (int i=1; i<=5 ; i++)
  {
  System.out.pln( int i=1 ; i<=5 ;; ++ )
  {
   S.O.pln (" Hello world");
  }

  for (int i=1 ; i<=5 ; i++)
  {
  S.O.p ( "i:" +i);
  }

for (int z=156 ;  z<=160 ; z++) //(int z=156 ; z<161 ; z++)
  {
 S.O.p ("z");
  }
  S.Op (" --- ");

for (int even=2 ; even<=10 ; even=even+2)
 {                                          O/p
System.out.println (even);                  2
}                                           4
system.out.println ("- -");                 6
                                            8
                                            10
for (int z=156; z<161 ; z++) // for (int z = 156 ; z<=160 ; z++)
 {
  S.O.p (z)                                 O/p
}                                           156
S.Op (" - - - ")                            157
                                            158
                                            159
```

```
for (int i=1 ; i<=9; i+=2)
{
s.o.p (i);
}
s.o.p (" --- ");
for (int z=3 ; z<=15 ; z=z+3);
{
s.o.p (z);
}
}
```

O/P
1
3
5
7
9

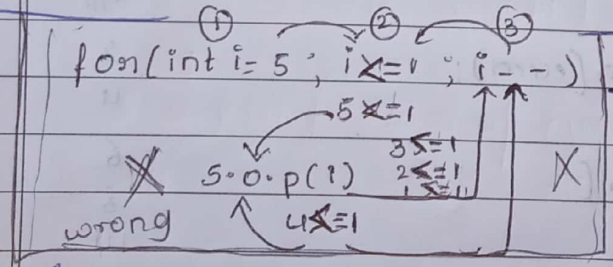
O/P
3
6
9
12
15

(Q) Write a program to print numbers from 1 to 10 in reverse order.

→
```
class Reverseorder
{
public static void main (String[] args)
{
for (int i=10 ; i>=1 ; i--)
{
System.out.println(i);
}
}
```

5>4>3>2>1

```
for(int i= 5 ; i<=1 ; i--)
```



→ Ex for infinite loop & NO O/P

infinite loop

```
for (int i=5 ; i>=0 ; ; i++)
{
s.o.p(i);
}
```

NO O/P

```
for (int i=5 ; i<=1 ; i--)
{
s.o.p(i);
}
```

Date: / /

Ex)
```
class ForLoopDemo
{
public static void main(String[] args)
{
for(int i=1; i<=10; i++)
  {
  if(i%2==0)
  {
  system.out.println(i); // 2==0
  }
}
  s.o.p("-----");

for(int i=1; i<=10; i++)
  {
  if(i%2==1)
    {
  system.out.println(i);
  }
}
  for(int i=1; i<=30; i++)
  {
  if(i%3==0) // if(i%5==0) multiples of 5
  {
  system.out.println(i);
  }
 }
}
}
```

O/P - 1
3
5
7
9

O/P : 3
6
9
12
15
18
21
24
30

Date: / /

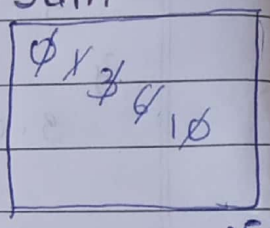
Q) print the sum of 'n' natural numbers.

→
```
Class NaturalNumbers
{
  psvm (String[] args)
  {
    int sum=0;
    for(int n=5;
    for (int i=1 ; i<=n ; i++)
    {
      sum = sum + i ;
    }
    s.o.p(sum)
  }
}
```

n=5

1+2+3+4+5

sum



Q) Write a program to find the sum of even numbers from 1 to 10

→
```
Class EvenNum
{
  psvm (String[] args)
  {
  for ( int i= 1 ; i<=10 ; i+2)
  {
      result = result + 1 ;
  }
  s.o.pln ("sum of Even nums :" + result);
  }
}
```

Date: 29/01/2020

While loop

* While loop is a looping statement which is used to execute a set of instructions until the condition is false.
* In other words while loop keeps on executing if the condition is true & stops when condition is false.

Syntax:-

```
While (condition)
    {
      Implementation
      - - - -
    }
```

Ex)

```
int i=1;
while(i<=5)
  {
  System.out.println(i);
    i++;
  }
```

1<=5	
2<=5	
3<=5	
4<=5	
5<=5	
6<=5 - false	

O/P

1
2
3
4
5

Ex)

```
int n=5
While(n>=1)
  {
  S.o.pln(n);
    n--;
  }
```

5>=1
4>=1
3>=1
2>=1
1>=1
0>=1 - F

O/P

5
4
3
2
1

Date: / /

do-While loop

* do While loop is a looping statement similar to While loop i.e it keeps on executing until the condition is false
* The difference between While and do While is
→ While checks the condition & then executes set of instructions
→ DoWhile loop executes set of instructions & then checks the condition.

Syntax:
```
do
{
  -- task
}
While (condition);
```

Ex)
```
int i=1;
do
{
  S.O.P(i);
  i++;
}
While(i<=5)
```
O/P: 1 2 3 4 5

Ex)
```
int x=5;
do
{
  S.O.Pln(x);
  x--;
}
While(x>=1);
```
O/P: 5 4 3 2 1

Date: / /

Difference between while & do while loop

	While loop	do w
*	While loop checks the condition first and then executes set of instructions	* do while loop executes set of instructions first and then checks the condition
*	If the condition is false While loop doesnot execute even once	* If the condition is false do while loop executes atleast once, even

*	Ex: int i=10
	while(i<=5)
	{ s.o.p(i)
	}
	o/p
	no o/p
		Ex):- int n=10;
		do
		{
		s.o.p(n)
		}
		while(n<=5);
		o/p
		10

Nested for loop

* Nested for loop is a combination of 1 for loop within another for loop.

Note: For 1 outer value, all the values of inner for loop gets executed.

Ex:

```
for(int i=1 ; i<=3 ; i++)
{
    s.o.p("Hi");
    for(j=5 ; j<=6 ; j++)
    {
        s.o.p(j+" "+j);
    }
    s.o.p("bye");
}
```

o/p :- Hi 25 36
15 26 bye
16 bye
bye Hi
Hi 35

Date: / /

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

for (i=1 ; i<=2; i++)
 {
System.out.println ("outer for loop start");

      for (j=1 ; j<=2 ; j++)
       {
          System.out.println ("Inside inner for loop
                i : " +i+ " j : " +j);
       }

    System.out.println ("outer for loop End");
   }
 }
}
```

output :- Javac nestedforloopdemo.java

Java nestedforloop

outer for loop start

inside inner for loop i:1 j:1

inside inner for loop i:1 j:2

Outer for loop End

outer for loop start

inside inner for loop i:2 j:1

inside. inner for loop i:2 j:2

outer for loop End

Date: / /

Q) Write a java program to print even numbers from 1 to 10 in reverse order using for loop, while loop, dowhile loop

10 8

for loop

```
for(i=10, i>=2 ; i-=2)
{
    s.o.p(i)
}
```

10 8

while

```
int n=10;
while(n>=2)
{
    s.o.p(n);
    n-=2
}
```

do while

```
int n=10;
do
{
    s.o.p(n);
    n-=2;
}
while(n>=2);
```

Date: / /

Object-Oriented Programming

Object : Object is a Real World Entity

* Anything which is present in the real world & has some physical existence can be termed as an object

Ex :- pen, humanbeing, car,

* The properties of an object generally categorized into 2 types
  1) States
  2) Behaviour
* States are the properties which is used to store some value
* Behaviour are the properties which is used to perform some task/action
* Ex : When we consider car as an object following are the states and behaviour

car

(storing data) States	Behaviour (performing task)
Brand	Start the car
color	stop the car
cost	shift gears
modelno	

* In programming perspective, object name is considered as class name, states are referred as variables, Behaviours are referred as methods/Functions

Class

30/1/2020

* Class is a blueprint of an object
* class also provides a platform to store the states & behaviour of a particular object
* class is a logical entity
* class has to predicate using a keyword called class
* Class can also act as a datatype

Syntax of Class :-

```
class classname
{
    //variables
    //methods/functions
    // constructors
    // Blocks
}
```

Scope of a class

Date: / /

Object Creation

* object creation is a process of storing or loading all the non static properties within the memory
* objects are created inside a memory location called as Heap Area
* We can create any number of objects for a specific class
* In simple words object is a copy or instante of a class

Syntax:- object creation

classname objectreferencename = new classname();

Ex: class pen
{
pen p1 = new pen();
}

Ex: class Boy
{
Boy b1 = new Boy();
}

* In onder to access nonstatic properties present within the object we make use of dot operaton (.)

Syntax: objectname.Variablename
(dot operator)

Q) How do we differentiate between a static variable & non static variable?

* If a variable is declared using static keyword we refer it as a static variable otherwise We refer it as non static variable

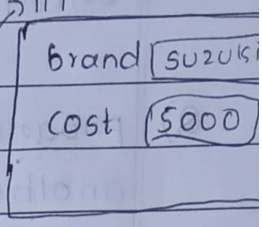
Variable
Static nonstatic

Ex:- Static int a=10; int a=5;

Date: / /

Q) Program to access non static variables present within the same class → class car

```
{
String Brand = "Suzuki"
int cost = 5000;
```




```
Public static void main (String [] args)
{
car c = new car()
```

constructor call

objectname

operator

classname

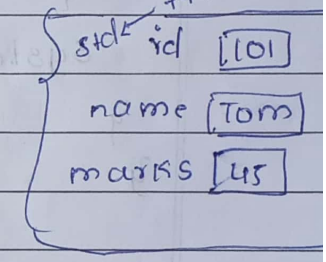
```
System.out.println(c.brand);
System.out.println(c.cost);
}
}
```

O/P :-

Suzuki

500

```
Class student
{
int id = 101;
String name = "Tom";
int marks = 45;
```



Heap Area

```
Public static void main (String [] args)
{
System.out.println("start");
Student std = new student();
System.out.println(std.id);
System.out.println(std.name);
System.out.println(std.marks);
System.out.println(std.id + " " + std.name + "
" std.marks)
System.out.println(std.name + " of id " + std.id + " has scored "
+ std.marks + " marks");
s.o.pln("end")
}
}
```

O/p :- start

101

TOM

45

101 TOM 45

TOM of id 101 has scored 45 marks end

Date: / /

Note: Non static properties either in the same class or another class can be access with the help of object creation

Q) program in order to access non static properties in another class.

→

```
class Employee
 {
    String name = "Jerry";
     int salary = 12000;

 }
        Creating an object of class employee
class Testemployee
 {
 public static void main (String[] args)
   {
   System.out.println(emp.name+" earns $"+emp.salary);
 }
}
```

Output

```
Javac Testemployee.java
java Testemployee
```

⇒ Jerry earns : $12000.00

Note: We can create any number of objects for a specific class & changes made to 1 object will not reflect to another

```
class pen
{
int cost=100; //nonstaticvariable

public static void main(String[] args)
 {
 // 2 copies or instances created for pen class
    Pen p1 = new pen();
    Pen p2 = new pen;
System.out.println(p1.cost + " " + p2.cost);
    p1.cost=300; // re-intializing cost to 300 pen obj p1
}
system.out.println(p1.cost + " " + p2.cost);

 }

}
```

Output

```
javar pen.java
java pen

100  100
300  100
```

Date: / /

Default value

* If a variable just declared & not initialized to any values, then the compiler will automatically initialize to its default value

Datatypes		Memory size		Default values
		Bytes	Bits	
Non-decimal	byte	1	8	0
Numeric values	short	2	16	
	int	4	32	
	long	8	64	
Decimal numeric values	float	4	32	0.0
	double	8	64	
true/false	boolean	1	8	false
Single character	char	2	16	unicode value '\u0'
Sequence of characters	String	X	X	empty space, null

Default values are applicable for both static & nonstatic variables

Example:

```
class DefaultValuesDemo
{
    int     a ;
    float   b ;
    boolean c ;
    char    j ;
    String  i ;

public static void main (String[] args )
{   DefaultValuesDemo dvd = new DefaultValuesDemo();
    System.out.println ( dvd.a);
    system.out.println (dvd.b);
    System.out.println (dvd.c);
    system.out.println (dvd.i);
    system.out.println (dvd.j);
}
}
```

O/p:

0

0.0

false

null

Date: / /

Q) Write a java program to store 2 student details wherein each student will have attributes called as name & Age.

```
class student
{
    int age=25;
    String name="Tom";

public static void main (String[] args)
    {
& student std = new stude( );

Syst.w
```

wrong

```
class student
{
String name;
int age;
```

stores default value

```
public static void main (String[] args)
{
String s1 student s1 = new student1( );
Stri      student s2 = new student( );

System.out.println(s1.name+" "+s1.age);
System.out.println(s2.name+" "+s2.age);

//reinitializing in s1 object
   s1.name = "Tom";
   s1.age  = 20;
//reinitializing in s2 object
   s2.name = "Jerry";
   s2.age  = 15;
```

```
System.out.println(s1.name+" is "+s1.age+"years old");
System.out.println(s2.name+" is "+s2.age+"years old");
}
}
```

output

```
null 0
null 0
Tom is 20 years old
Jerry is 15 years old
```

31/1/2020

Methods / Functions

* Method is a block of code which is used to perform a specific task
* In other words, method is a set of instructions to represent the behaviour of an object
* Methods can also be referred as Functions

Different ways of writing a method are as follows

i) Method without arguments without return statements
ii) Method with arguments without return statement
iii) Method without arguments with return statement
iv) Method with arguments with return statements

Syntax: Accessspecifier returntype methodName( ) (optional arguments / inputs)

```
{
    statement 1;
    ....
    statement n;
    return;
}
```

Note: A method gets executed only when we call it or invoke it

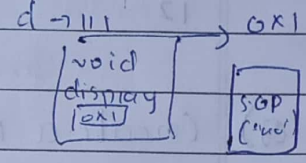
Date: / /

Syntax: objectname.methodName();

Q) Program to demonstrate a method without arguments without return statements

```
class Demo
{
    void display()
    {
        System.out.println("Helloworld");
    }

    public static void main
    {
        S.o.pln("start")
        Demo d = new Demo();
        ( d.display(); ) → (calling Method)
        S.o.pln("End");
    }
```

Method defining



O/P

Start

Helloworld

End

Q) Program Method with Arguments & without return Statements

```
class Addition
{
    //method with arguments & without return statement
    void add(int a, int b) // Arguments
    {
        System.out.println(a+b);
    }
```

Date: / /

```
public static void main(String[] args)
  {
     Addition obj = new Addition();
          obj.add(10,20) // paramters (int type)
          obj.add(5,7);
  }
}
```

O/P

30

12

Q) Create a method called as multiply which accept the 3 arguments and prints the products of those 3 numbers

→

```
class multiply
{
Syst.  return void multi(int a, int b, int c)
        {
        system.out.println(a*b*c);
        }

public static void main(String[] args)
   {
      Multiply obj = new multiply;
          obj.multi(10,20,30)
      }
  }
}
```

O/P

6000

```
PSVM ( -- )
{
S.op ("start")
Test t = new Test ();
int result = t.display();
S.O.p [result];
S.O.p (t.display());
S.O.p ("end");
}
```

```
class Test
{
int display ()
{
return 10;
}
}
```

(1) (2) (3) (4) (5)

(*) Method with No Agruments & with return statements

```
class Test
{
// Method with no arguments & with return statem
String display ()
{
return "Hello";
}

public static void main (String[] args)
{
System.out.println ("start");
test t = new test ();
t.test String result = t.display();
S.O.pln (result);
S.O.pln (t.display());
S.O.pln ("End");
}
}
```

O/P

start
Hello
Hello
End

Method with arguments & with return statements

```
class calculation
{
    int test (int a)
    {
        return a*a;
    }

public static void main (String[] args)
    {
        Calculation c = new Calculation
            c.calculation c
        int res = c.test (2)   pass parameter
            System.out.println (res);
            System.out.println ("5 power 5 " + c.test(2));
        }
    }
```

O/P

4

25

Which Return type to use

→

```
int test()
{
    return 10;
}
```

```
String test()
{
    return "Dinga";
}
```

```
boolean test()
{
    return 10<5 false;
}
```

```
String test()
{
    String name = "tom";
    return "Name;" +nam;
}
```

O/P

Name : tom

String cannot concatinate with int

```
double test()
{
return 10.5;
}
```

```
String test (int age)
{
return "Age:" +age;
}
```

age → bc3 (i++)

bc3 (i++)

```
int test()
{
 int a=5+5;
return a;
}
```

```
String test (double height)
{
return " " +height;
}
```

Date: / /

Write a pgm to check even or odd

```
class solution {
    string checkEvenoroddd(int num)
        {
          if (num%2==0)
          {
            return num+" is a Even num";
          }
        else
        {
         return num+" is a odd num";
        }
    }

public static void main(String[] args)
    Solution sol= new Solution();
    system.out.println(sol.checkEvenoroddd(n));
```

1/02/2020

Method Overloading

- Method overloading is a process of having multiple methods with the same name but with difference in arguments.
- In other words, in a class having multiple methods or functions with samename but change in the argument
- In order to achieve method overloading we have to satisfy atleast one of the following 3 rules

Rule1 There should be a change in the number of arguments i.e length of the arguments

Rule2 There should be a change in the datatype of the arguments

Rule3 There should be a change in the order or the sequence of the datatypes

Note Variable names might be same or different

Note Compile time polymorphism can be achieved with the help of method Overloading

Date: / /

Q1)
```
class MO
{
 void display (int a)
 { s.o.pln(a);
 }

 void display (int a double b)
 {
    s.o.pln (a+ "\t" +b);
 }

 void display (string a, string b)
 {
    s.o.p (a+ "\t" +b);

 }

 void display (int a, string b )
 {
    s.o.p (a+ "\t" +b);

 }

 void display (string a )
 {

    s.o.p(a);

 }
}
```

Date: / /

Another class

```
class TestMO
{
public static void main (String[] args)
  {
     MO m = new MO();
      m.display(10);
      m.display ("Hello");
      m.display (10, "Hello");
      m.display ("Hi ", 50);
       m.display(10, 10.5);
  }
}
```

Output

```
10
Hello
10 Hello
Hi 50
10 10.5
```

Note: During methodoverloading returntype can be same or different

(*) ✓

(*) ✓ x We can Overload static methods and non static methods as well

(*) (✓) * Even main methods can be overloaded but the execution will always begin from the method which accepts a String[] (String array) as the datatypes

```
P S VM (double[] a)
  {
  }
}
```

```
psvm (string[] a) ⟶ execution starts from this
{
}

psvm (int[] a)
{
}
```

* 1 Q) In java we cannot have nested methods but a method can call on invoke another method.

Static keyword

* Static is a keyword which can be used for Variables & methods

* Static properties will have 1 copy per class

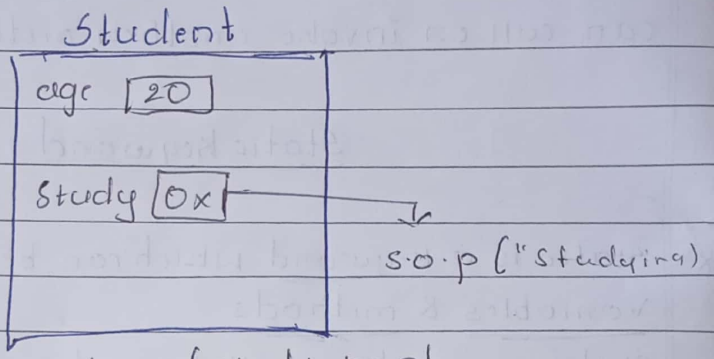
* All the static properties are loaded into memory location called as class Area / static pool

* In order to define a static variable / static method we have to make use of static keyword

* If we have any static properties then we should always access it with the help of classname

```
class student
{
  static int age = 20;
  static void study()
  {
  system.out.println("studying");
  }
```

```
public static void main (String [] args)
  {
  System.out.println ("start");
  System.out.println ("student.age");
                      ((classname.v)
method ← student.study();
    S.O.pln ("end");
  }
```



Output
start
20
studying
end.

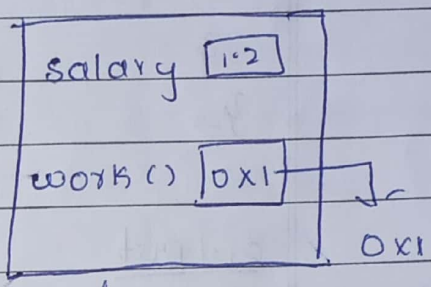
Note: static properties can be accessed directly or with the help of classname within the sameclass

```
Class Employee
{
 static double salary=1.2;
 static void work()
     {
     S.O.pln ("Employee is working");
    }
```

```
public static void main (String[] args)
  {
  System.out.println (Employee.salary);
      Employee.works();
                          //className.salary -> Employee.salary
  System.out.println (salary);
  work(); // internally its like className.works() -> Employee.work()
  }
}
```

output

```
Emp sal : 1.2
Employee is working
1.2
Employee is working
```



Class Area / static s.o.p pool ("E is w");

class

car.java

```
class car
{
static int price=12000;
static void start()
  {
  System.out.println("car started");
  }
}
```

Testcar.java

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

```
System.out.println (can.price);
car.start();

/* ERROR
    S.O.pln(price);
    start();
internally its Testcan.price & Testcar.start()
wherein Testcan has no variable called price
& no method called as start() so ERROR */

}
}
```

Output

```
javac Testcan.java
java Testcan
  12000
  car started
```

```
class pen
{                    static int cost = 50; // static variable
public static void main (String[] args) //
{
system.out.println(cost); // pen.cost
    cost = 70;        // pen.cost = 70;
system.out.println (pen.cost); //tp
}
}
```

Output

```
50
70
```

Pen

cost	~~50~~ 70

Print or RC.	Accessing in same class	Accessing outside the class	Memory location	No of copies
Static Variable & Method	(1) Directly or (2) with ClassName	(1) with the ClassName	Class Area / static pool	only 1 copy
Non-static properties i.e variables Methods	object Creation	Object Creation	Heap Area	Depends on No of objects created

```
class person
{
    int age=10;
}
person p1 = new person();
  "    p2 = new   "  ;
  "    p3 = new   "  ;
```

3 copies / 3 objects / Instances

p1   p2   p3

[age=10]   [age=10]   [age=10]

Date: / /

Q) Can we access static properties with class Name

=> yes

Q) Can we access non static variables with the help of objectReferenceName

=> yes

Q) Can we access static properties with the help of objectReferenceName,

=> yes. Each & every object will implicitly point to the static pool

Q) Can we access non static properties with the help of classname?

=> NO because nonstatic properties are loaded only when we create an object

(*)

```
class pen
{
static int a=10;    // static variable
       int b=20;    // non static variable

public static void main(String[] args)
   {
      system.out.println(pen.a);
        //system.out.println(pen.b);
       pen p=new pen();
  system.out.println(p.b); // accessing both static & non static
  system.out.println(p.a); // variable using object reference name
   }
}
```

O/P

10

20

10

paperg

Date: /

	ClassName	Objectreference nam i.e (After obj creatn)
Static properties (variables & Methods)	✓	✓
non-static properties (V & M)	✗	✓

Date: / /

Java Development kit (JDK)

* JDK stands for Java Development kit
* JDK is a s/w which contains all the resources for developing & executing java pgms

Java Runtime Environment (JRE)

* JRE stands for Java Runtime Environment
* JRE is a s/w which provides a platform for executing java pgms
* It is possible to install only JRE but if only JRE is installed we can execute the pgms but cannot develop it

JIT (Just In Time)

* JIT stands for Just in Time Compiler
* JIT is responsible for converting the bytecode into machine level language.

Class Loader

* Class loader is the one who loads the class from secondary storage to the executable area

Interpreter

* Interpreter is used to execute java pgm line by line
* JVM stands for V

JVM

* JVM stands for Java virtual Machine
* JVM is the smartness or manager or the implementar of JRE

Date: / /

JVM Architecture

* The memory location are categorized into follows

(1) **Heap Area**

(2) - All the objects are stored inside heap area i.e non static properties

(2) **Class Area / static pool**

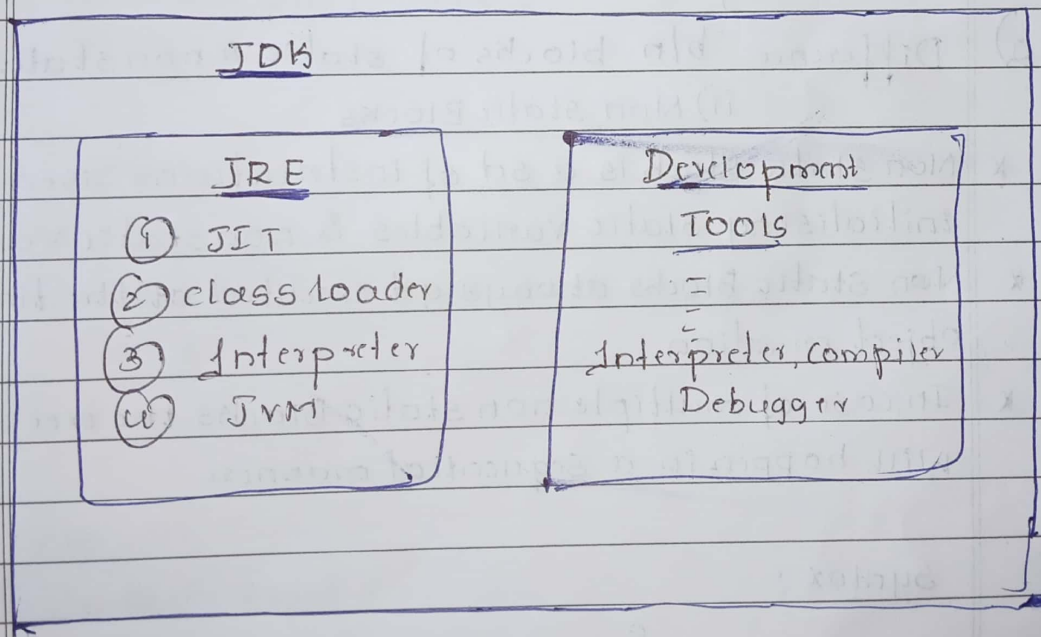
All the static properties are loaded inside class area

(3) **Method Area**

The method def / implementation is always stored inside method area

(4) **stacks**

The execution of a java pgm always happens inside the stacks



- Heap Area
- class Area
- Method Area
- stacks

Date: / /

Blocks

* Blocks are a set of instructions used for initialization
* Blocks are generally categorized into 2 types
  i) Static Blocks
  ii) Nonstatic Blocks

i) Static Blocks

* Static Blocks is a set of instructions used for initializing static variable
* Static Blocks gets executed at the time of class loading or just before main method.
* We can have multiple static blocks & execution will happen in sequential manner.

Syntax:
```
static
{
----
}
```

* Q) Difference b/n blocks of static & nonstatic.

ii) Non-static Blocks

* Non static Block is a set of instructions used for initialising static variables & nonstatic variable
* Non static Blocks always get executed at the time of object creation.
* In case of multiple non static Blocks the execution will happen in a sequential manner.

Syntax :

```
{
---
--
}
```
Blocks

(Blocks without static keyword is nonstatic Block)

Q) package com;

```
public class Demo
 {
  static
  {
   System.out.println("In static block-1");
  }

 static
  {
   System.out.println ("In static block-2");
  }

 public static void main (String[] args)
    {
     S.O.pln ("Hello");
    }

 static
  {
  S.O.P ("In static block-3);
    }
}
```

O/P

In static Block 1

" " 2

" " 3

Hello

8) package com;

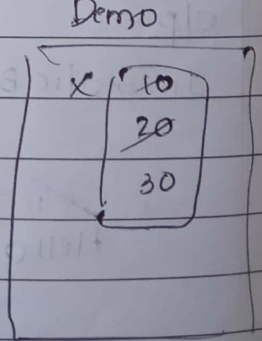
public class Demo5

```
static int x=10;

static
{
system.out.println("Initializing to 20");
   x=20;
}

public static void main (String[] args)
{
  S.O.pln ("start");
   S.O.pln (x)
   S.O.p ln ("End");
}

static
{
S.O.pln("Initializing to 30");
  x=30;
}
}
```

output

Initializing to 20
" " 30
Start
30
End

I -- 20
I -- 30
S
30



Class Area / static pool

Date: / /

Example for non static Blocks

(execute only when obj created)

```
package com;

public class Test {
    {
        System.out.println("In Non-static Block-1");
    }

    public static void main (String [] args)
    {
        S.o.pln ("start");
        Test t = new Test(); // obj creatn
        S.o.pln ("End");
    }

    {
        S.o.pln ("In non-static Block-2");
    }
}
```

non static block

non static Block

Output : start
In non static Block-1
In Non static Block-2
start End

3)
```
package com;
public class student
{
    int age;
    {
        S.o.pln ("Initializing to 20");
        age=20;
    }
```

Date: / /

```
public static void main (String[] args)
    {
      system.out.println ("start");
        student s = new student();
        s.o.pln (" Age :" +s.age);
        s.o.pln ("end");
    }

{
  s.o.pln (" initializing to30");
     age=30;
  }
}
```

output

start
Initializing to 20
Initializing to 30
Age:30
end

*) static variable ; static blocks
*) nonstatic variable, nonstatic block
↓
Q) public class car {
    static int x=10; //1
     int y=20; //4

    static //2
    {
    s.o.pln (" in SB-1");
    }

```
{ //5
  S.O.Pln("In NSB-1");
}

prSM(String[] args) //3
{
  S.O.pln("Hello");
  S.O.pln(car.x);
    car C = new car();
    S.O.p in (C.4);
}
```

Output

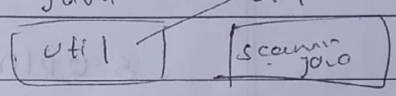
in SB-1

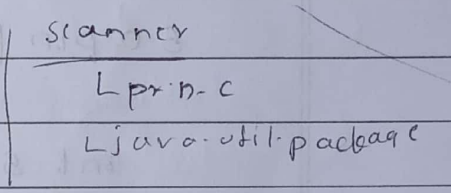
Hello

10

in NSB-1

20





Scanner

- Scanner is a predefined class present in java.util package
- Scanner class is used to dynamically accept the inputs from the user.

Rules for using Scanner class

i) Create an object of scanner class

ii) pass system.in to the constructor call of the scanner object (to accept input)

syntax: Scanner scan = new Scanner(system.in);

iii) import the scanner class using below syntax

import java.util.Scanner

* Make use of predefined methods in order to accept specific types of inputs

```
package org;
import java.util.Scanner;

public class Demo{
  public static void main(String[] args)
  {
    S.o.pln("start");
    Scanner scan = new Scanner(System.in);

    S.o.pln("enter a:");
     int a = scan.nextInt();
     S.o.pln("value of a is:"+a);

    S.o.pln("Enter b:");
     int b= scan.nextInt();
    S.o.pln("value of b is" +b);

      int sum=a+b;
       S.o.pln("sum:"+sum);
       S.o.pln("End");
  }
}
```

O/P

start
Enter a
10
Value of a is : 10
Enter b
20
value of b is 20
Sum:30
End

1) byte - nextByte()
2) short - nextShort()
3) Int - nextInt()
4) long - nextLong()
5) float - nextFloat()
6) Double - nextDouble()
7) boolean - nextBoolean()
8) String - next() or
nextLine()

9) Char → next().charAt(0) index value at 0th posn
"Java".charAt(0)
'J'

5|2|2020

Constructors :-

* Constructors are special set of instructions used for initialisation & instantiation
* Initialisation means assiging the values
* Instantiation means object creation
* Constructor name & class Name should always be same
* Constructors will never have returntype.
* Constructors get executed at the time of object creation

Syntax :

optional Inputs / Arguments

```
AccessSpecifier ClassName ( )
{
    ---
    ---
    ---
}
```

Scope of constructor

* Constructors are generally classified into 2 types
  i) Default constructor
  ii) Custom/user defined constructor

Default Constructor

* If a class doesnot have any explicit constructor, then the compiler would automatically generate a constructor & therefore this constructor referred as default constructor
* Default constructor neither accepts any arguments nor have any implementation

Custom Constructor

* If a constructor is explicitly defined by a user or programmer, then it is referred as custom constructor
* Custom constructor can be further categorized into 2 types
  i) Non parameterized Constructor
  ii) parameterized constructor

Note If there is default constructor, custom constructor cannot be present & vice versa i.e if there is custom castructor, default constructor cannot be present

Q)
```
public class pen
{          //non parameterized or non argument
pen()                custom constructor
{
s.o.pln("In pen class constructor");
}
```

Date: / /

```
psvm (String[] args)
{
  S.o.pln("start");
  pen p = new pen();
  S.o.pln("end");
  }
}
```

O/P

start

In pen constructor

End

Example for parameterized / with argument construction

```
class can
{
   can(int a)
   {
   S.op(a);
   }
```

so

=> psvM(- - - -)

→ sop("start");

can c = new can(50);

Sop("End");

Constructor call

↓

i.e invoking a constructor

O/p

start

50

End

Date: 6/9/2020

Global or Member Variable
OR
Local Variables

* If a variable is declared within a class directly, then we refer it as global variable or member variable
* Member Variable can be access globally i.e everywhere
* If a variable is declared within a specific scope such as method, constructor etc we refer it as local variable.
* Local variables are accessible only within a specific scope

Note Default values are applicable only for global/Member Variables. i.e either static or nonstatic.

Q)
```
Public class Demo {
    int a;    // global or Member variable

 void display (int b) // b & c are local variable
  {
   int c=20;
   System.out.pln("a" "+b+" "+c);
  }

 Demo(int x)  // x & y are local variable
  {
   int y=40;
   S.O.pln ("a" "+x+" "+y);
  }

 public static void main (String[] args)
    {                                    O/P
       Demo obj = new Demo(50);          0 50 40
       obj.display(30);                  0 30 20
       S.O.pln(obj.a);                   0
    }
}
```

this Keyword

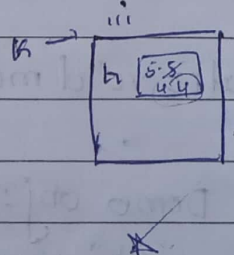
* this is a keyword which is used to point to the current object
* In java it is possible to have member variable & local variable name same ∴ Always the local variables will dominate over member variables
* Hence in order to point to the current object we make use of this keyword

Q) Public class Kangroo
{
double height = 5.5;

void display( )
{
double height = 4.4;
S.o.pln (height);
S.o.pln (this.height);
}

psvm (String [] args)
{
Kangroo k = new Kangroo();
k.display();
}
}

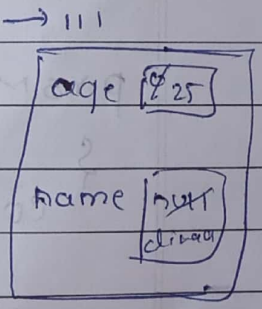
output
4.4
5.5



Date: / /

Q) Class person

```
{
  int age;
  String name;

person (int age, String name)
{                25          "dinga"
    this.age = age;
    this.name = name;
}                    "dinga"

}

PVSM (----)
{
  person p = new Person (25, "dinga");
    S.o.p (p.age + " " + p.name);
           (25)        dinga
}
```



Q) Write a pgm to print follow below scenario

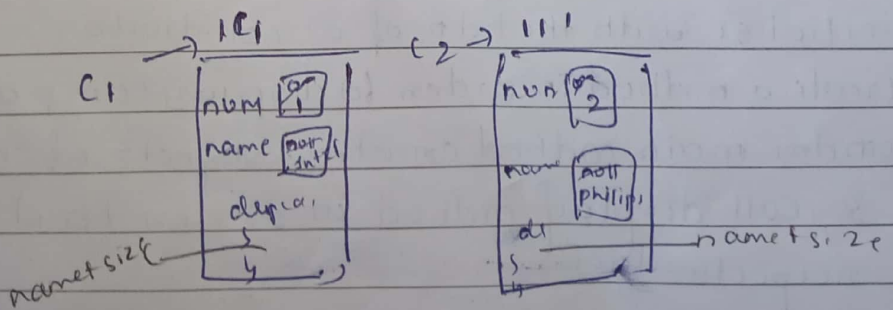
(i) create a class called as company
(ii) Define 2 variables as company name, no of employee
(iii) initialise with the help of a constructor
(iv) create a method in order to display company details
(v) under main method create 2 objects of employee & call display method in order to display respective details

Date: / /

```
public class company
{
    int num ;
    String name;

    company (num int num, string name)
    {
        this.num=num;
        this.name=name;

    }

    void display ( )
    {
        s.o.pln (" Employee name is "+name+"
                no of employee : "+num);
    }

prsM (string[] args)
{
    company c1 = new company (2, "Intel")
        c1.display( );
    company c2= new company ( 2, "philips")
        c2.display( );

}
```

this.name

this.size()



Date: / /

8) Public class Employee

```
{
  int empId;
  string empName;
  double empSalary;

Employee(int empId, string empName, double emp
                                          salary)
  {
    this.empId   = empId;
    this.empName = empName;
    this.empSalary = empSalary;
  }

void display()
  {
S.O.pln("Employee Id : "+this.empId);
S.O.pln("Employee Name : "+this.empName);
S.O.pln("Employee Salary : "+this.empSalary);
  }

PVSM(String[] args)
  {
    Employee e1 = new Employee(1, "A", 1.2);
    Employee e2 = new Employee(2, "B", 2.2);
    Employee e3 = new Employee(3, "C", 3.3);
    e1.display();
    e2.display();
    e3.display();
  }
}
```

OLD :-

Employee Details

Emp Id : 1	3
Name : A	C
Salary : 1.2	33
: 2	
: B	
: 2.2	

S.O.pln("Emply Details");
S.O.pln("---");

8)
```
import java.util.Scanner;

public class car {
    String brand;

    car (string brand) {
        this.brand = brand;
    }

prsM (String [] args) {

Scanner s = new Scanner (system.in);
s.o.pln ("Enter brand");
string brand = s.next();
       b

car c = new car (brand);
                 b
s.o.pln ("Brand Name is " + c.brand);
```

O/P

Enter brand

BMW

Brand Name is BMW

Date: / /

Constructor Overloading

* Constructor Overloading defining more than 1 Constructor in a class known as constructor Overloading
* Constructor should defer in parameter list either in parameter type or parameter length
* Any 2 constructors in the class shouldnot match the parameter type
* The overloaded are called based on arguments / parameter type

Q) Constructor Overloading

```
class program
{
int x;
int y;

program(int arg1)
{
  x = arg1;
}

program(double arg2)
{
  y = arg2;
}

program (int arg1, double arg2)
  {
    x = arg1;
    y = arg2;
  }
```

Date: / /

```
void display()
 {
 S.O.p("X value" + x);
 S.O.p ("y valor" +y);
 }
}

class Mainclass
 {
  psvm (String[] args)
  {
 program p1 = new program(12);
    p1.display();
 program p2 = new program(3.4);
    p2.display();
 program p3 = new program(12, 3.4);
     p3.display();
  }
}
```

8/2/2020 **Constructor chaining :-**

* Constructor chaining is a process of 1 constructor calling another constructor.
* Constructor chaining can be achieved with the help of this calling statement (this())
* In order to achieve constructor chaining we need to have constructor Overloading as well
* this() calling statement should be the first executable line within the constructor
* this() calling statement is used to invoke another constructor in the same class

Date: / /

Ex)
```
class Demo
{
 Demo()
 {
  this(10);
  S.O.pln(1);
 }
Demo(int a)
 {
 S.O.pln(2);
 }
prSM(string[] args)
{
 S.O.pln("start");
Demo d = new Demo();
  S.O.pln("End");
 }
}
```

O/P : start
2
1
End

Imp* NOTE : Recursive chaining is not possible in java
∴ If there are n constructors we can have maximum of n-1 casting statements

Ex :
```
class Demo
{
 Demo()
 {
  this(10);
  S.O.p(1);
 }
Demo(int a)
 {
  this();
  S.O.p(2);
}
```

```
psvm(String[] args)
{
Demo d = new Demo();
}
}
```

o/p
infinite

Ex:

```
Package chaining
  class student
{
Student(int id)
 {
 s.o.pln("Id:"+id);
 }
 student(int age, int marks)
 {
  this("Tom");
  s.o.pln("Age:"+age+" Marks:"+marks);
 }
  psvm(---)
{
student s = new student(25,67);
}
}
  Student(String name)
    {
      this(101);
      s.o.pln("Name:"+name);
    }
```

o/p

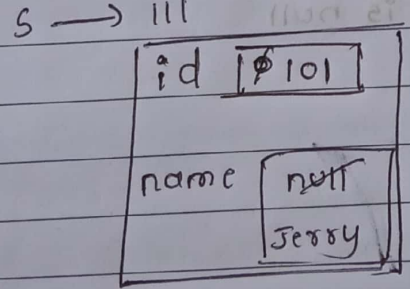
Id:101
Name:Tom
Age:25
Marks:67

Ex2)

```
package chaining;
public class student
{
  int id;
String name;
student (int id)
{
this.id=id; // current object id=101;
}
 student (int id, string name)
  {
    this(id); // this(101);
    this.name=name;
  }
 psvm(---)
  {
    student s = new student(101, "Jerry");
      system.out.println(s.id+" "+s.name);
  }
}
```

O/P

101 Jerry



Date: 11/2/2020

Datatypes

* Datatypes are an indication on specification of what type of data is assigned to a variable into a
* Datatypes are typically categorized into
  i) primitive ___ class keyword
  ii) Non-primitive ___ class

i) Primitive datatypes

If a keyword is behaving as a datatype, then it is a primitive data type

* All primitive data types have fixed size
* Java supports only 8 primitive datatypes, they are
  i) int
  ii) byte
  iii) short
  iv) long
  v) float
  vi) double
  vii) boolean
  viii) char

ii) Non primitive data types

* If a class is behaving as a datatype, then it is a Non primitive datatype
* Non primitive datatype does not have fixed size
* We can have any number of non primitive datatype
* And best example is String

Note: The default value for any non primitive or class type or interface type is null

Date: / /

Arrays

* Array is a container in order to store group of data objects
* Arrays are always fixed size
* Arrays are of homogenous type i.e only similar type of data can be stored
* Arrays are indexed based & the indexed position always starts from 0

(1) Array Declaration

Syntax : datatype[] arrayName;

or

datatype arrayName[];

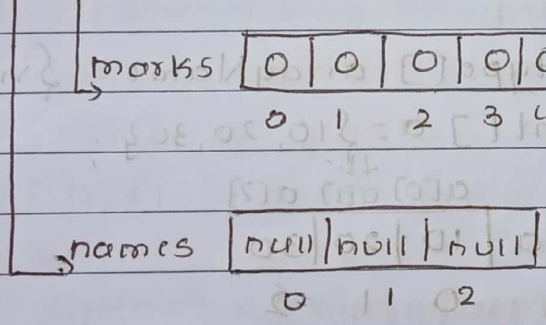
or

datatype [] arrayName;

(2) Array Creation

Syntax : arrayName = new datatype[size];

```
marks = new int[5];
names = new String[3];
```



(3) Array Declaration & creation together

Array Declaration

Syntax : datatype[] arrayName

datatype[] arrayName = new datatype[size]

Array Creation

double [] Salary = new double[3];

Salary	0.0	10.56	12.34
	0	1	2

iv) Array Initialisation

Syntax :- arrayName[index] = value;

Salary[1] = 10.56;

Salary[2] = 12.34;

int [] a = new int[3];

	a[0]	a[1]	a[2]
a	ø500	ø300	ø100

a[0] = 500;

a[1] = 300;

a[2] = 100;

v) Array Initialisation Directly

Syntax :- datatype[] arrayName = {v1, v2, v3 ...};

int [] a = {10, 20, 30};

	a[0]	a[1]	a[2]
a	10	20	30
	0	1	2

Q)

```
Package com;

Public class Demo:
{
Public static void main (String[] args)
  {
    // Array Declaration
    int[] a;

    // Array creation
    a=new int[3];

    // printing Array Elements
    System.out.println(a[0]);
    System.out.println(a[1]);
    System.out.println(a[2]);

    System.out.println("====");

    // Initialising Array using Index function s
    a[0]=10;
    a[1]=20;
    a[2]=30;

    System.out.println(a[0]);
    System.out.println(a[1]);
    System.out.println(a[2]);
  }
}
```

→ // int[] a= new int[3];

O/P:-

```
0
0
0
====
10
20
30
```

8) Write a java pgm to traverse the array elements both in forward direction & reverse direction

i<=2 => i<=a.length-1 (3)

pu i<3 => i<a.length

int[] a= int[] a={10,20,30};

a	10	20	30
	0	1	2

Forward traverse

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

0<3, 1<3, 2<3, 3<3 ×

forward

a[0] 10
a[1] 20
a[2] 30

Reverse traverse

a	a[0] 10	a[1] 20	a[2] 30
	0	1	2

```
for(int i=a.length-1; i>=0; i--)
{
    S.o.p(a[i]);
}
```

2>=0, 1>=0, 0>=0, -1>=0 ×

30
20
10

Date: / /

```
public class test
{
psvm (String[] args)
 {
   int[] a= {10, 20, 30};

   s.o.pln(a[60]);
   s.o.pln(a[20]);
   s.o.pln(a[2]);

   s.o.pln("------");

  {int len=a.length;
   { s.o.pln(len);
                OR//
  { s.o.pln("Array Length:"+a.length);
   }
  }
```

o/p

```
10
20
30
------
3
Array Length:3
```

Q) Write a java program to find sum & Average of array elements

```
publi class Demo
 {
public static void main (string[] args)
   {
      int sum=0;
         int[] a= {10, 20, 30};

    for(int i=0; i<a.length; i++)
      {
       sum = sum+a[i]; // sum+=a[i];
      }

    S.O.pln("sum:"+sum);

      int avg= sum/a.length;
       S.O.pln("Avg :"+avg);
   }
}
```

Sum: 64
Avg = 21

Assignment

Q) Write a program to dynamically accept array elements & find the no of occurances of a specific element

8)
```
import java.util.Scanner;
class Demo
{
public static void main (String[] args)
  {
    Scanner Scan = new Scanner (System.in);

System.out.println ("Enter no of elements to be
                     Inserted");
  int size = Scan.nextInt();

  int[] a = new Int [size];
```

```
System.out.println("Enter "+a.length+" Elements");

for(int i=0; i<a.length ; i++)
  {
    a[i].scan.nextInt();
  }

System.out.println("Array elements are as follows");

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

O/P

```
Enter No of elements to be Inserted
 5

Enter 5 elements
10
20
30
40
50

Array Elements are as follows
10
20
30
40
50
```

No of Occurence

```
System.out.println("Enter element to be searched");
    int ele = scan.nextInt();

    int count=0;

for(int i=0 ; i<a.length; i++)
    {
      if(ele==a[i])
       {
         count++;
      }
   }
System.out.println(ele+" has occured "+count
                              +" times");
  }
}
```

O/P

Enter element to be searched
10
10 has occured 3 times

```
int count=0;
int[] a = {10, 20, 30, 10};
int ele=10;
for(int i=0 ; i<a.length; i++)
```

a	10	20	30	10
	0	1	2	3

0<4
1<4
2<4
3<4

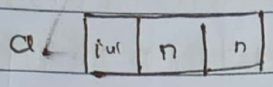
a[i]

```
if(a[i]==ele)
    a(0)=:.
{
  count++
}
```

Date: 12/2/2020

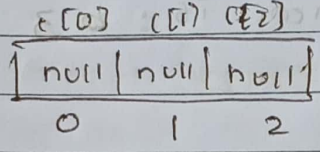
Storing objects Inside an Array

```
int[] a = new int[3];
```



a can store multiple values of int

```
Class Car
{
    Car[] c;
    c = new Car[3];
}
```




```
class Car
{
  int cost;
  String brand;

per   Car(int cost, String brand)
      {
          this.cost = cost;
          this.brand = brand;
      }

public static void main(String[] args)
  {
    Car c1 = new Car("BMW", 100);
    Car c2 = new Car("Suzuki", 200);

    or Car[] c = new Car[2]
        { c[0] = c1;
          c[1] = c2;
    same { //Car[] c = {c1, c2};
```

```
for(int i=0 ; i<c.length; i++)
{
    System.out.println(c[i]);
    System.out.println(c[i].brand+" "+c[i]
                                    .cost);
    System.out.println("====");
}
}
```

O/P

org.car@7852e922
BMW 100
====
org.car@4e25154f
Suzuki 200

c[0] c[1]

c | 111 c1 | c2 222

c[0] = c1 ; → 111

c[1] = c2 ; → 222

array name = c

for(int i=0 ; i<c.length ; i++)

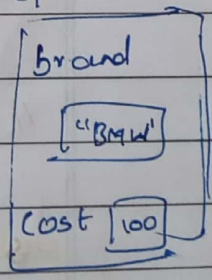
0<2

sop(c[i].brand+" "+c[i].cost);

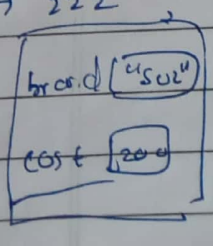
111.brand 111.cost

BMW

c1 → 111



c2 → 222



Date: / /

```
class student
{
  int id;
 String name;
  int marks;

  student (int id, string name, int marks)
{
   this.id=id;
   this.name=name;
   this.marks=marks;
}

public static void main (string[] args)
{
 student s1 = new student (1, "Bhagya", 100);
 student s2 = new student (2, "Rekha", 200);
 student s3 = new student (3, "Nivi", 300);

 student[]s= new
  student [] s = new student[3];

// student[] s = {s1, s2, s3}

s.o.pln for (int i=0 ; i<s.length ; i++)
      {
          s.o.pln (s[i]);
 s.o.pln(s.id+" "+s.name+" "+s.marks)
      }

}

}
```

qlp

Date: / /

//

Off

```
for (int i=0 ; i< std.length ; i++)
    {
      //student s = std[i]
      //s.o.pln(s.id+" "+s.name+" "+s.marks);
s=o.pln(std[i].name+" "+std[i].id+" "+std[i]
                                    .marks);
    }
```

O/P

org.123

1 Bhagya 100

org.222

2 Rekha 200

org.333

3 Nivi 300

Date: / /

Inheritance :-

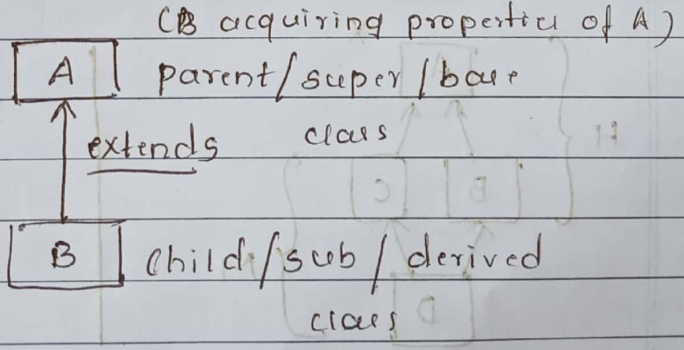
* Inheritance is a process wherein one class is acquiring properties from another class
* A class which shares the property is called a superclass / parent class / base class
* A class which acquires the properties is referred as subclass / child class / derived class
* Inheritance in java is achieved using extends keyword
* Inheritance is also referred as IS-A Relationship
* In java only variables & methods are Inherited where as blocks & constructors are not Inherited

Different ways of Inheritance are as follows

1) Single (level) Inheritance
2) Multilevel Inheritance
3) Hierarchical Inheritance
4) Multiple Inheritance
5) Hybrid Inheritance

(i) Single level Inheritance

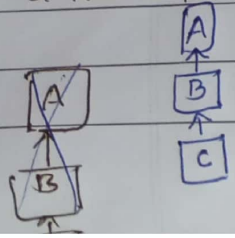
A combination of 1 super class & 1 sub class is called as single level Inheritance



(ii) Multilevel Inheritance

A combination of Multiple super classes & multiple subclasses in a sequential order

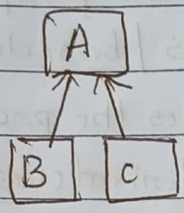
B is a superclass of & subclass of



Date: / /

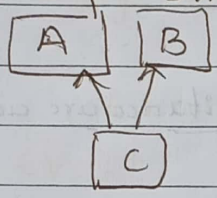
(iii) **Hierarchial Inheritance**

A combn of 1 superclass & multiple subclass is called Hierarchal Inheritance



(iv) **Multiple Inheritance**

A combn of 1 subclass having & Multiple superclass is called Multiple Inheritance

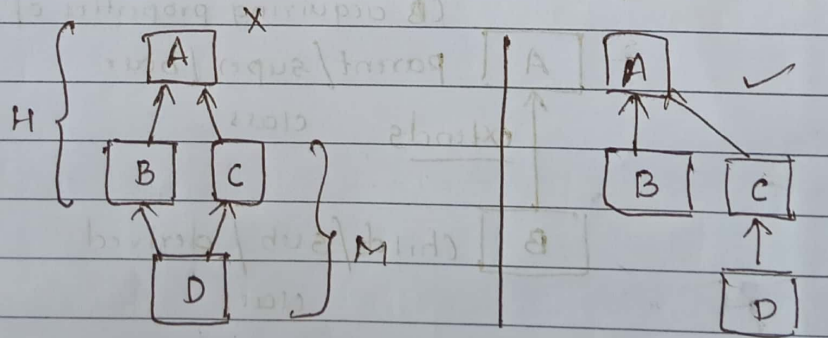


(v) **Hybrid Inheritance**

It is a combination of 2 or more types of Inheritance

(x) In java hybrid inheritance is supported upto some extent not completely

**Note : Java DoesNot Support Multiple Inheritance



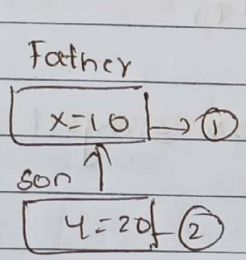
alt shift - f double click on name

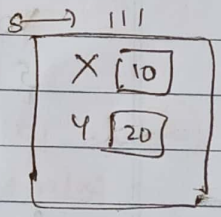
papergrid

Date: / /

Example of Single level Inheritance

```
class Father
{
  int x=10;
}

class son extends Father
{
  int y=20;

public static void main (String[] args)
  {
    son s = new son();
    s.o.p(s.x+" "+s.y);
  }
}
```





son is implicitly accessing property of father

Example for Multilevel Inheritance

4 classes

1. University.java

```
Public class University
{
  String universityName = "VTU";

void no of colleges()
      {
        s.o.pln("No of colleges are 120");
  }
```

Date: / /

college.java

```
public class college extends University
 {
   String collegeName = "Jspiders";

 void noofstudents()
   {
    s.o.pln("No of students are 4000").
   }
}


publicclass Department extends college
 {
   String departmentName = "CSE";

  Void favsubject()
  {
    s.o.pln("Fav subject is JAVA");
   }
}


public class Solution
 {
 public static void main(String[] args)
  {
    Department d = new Department();

 s.o.pln("university Name: "+d.UniversityName);
 s.o.pln("college Name : "+d.collegeName)
 s.o.pln("Department Name :" d.department
                              Name);
```

Date: / /

```
S.O.pln(" - ---- - ");

 d.noofcollega();
d.no of students();
d.favsubject();

}
}
```

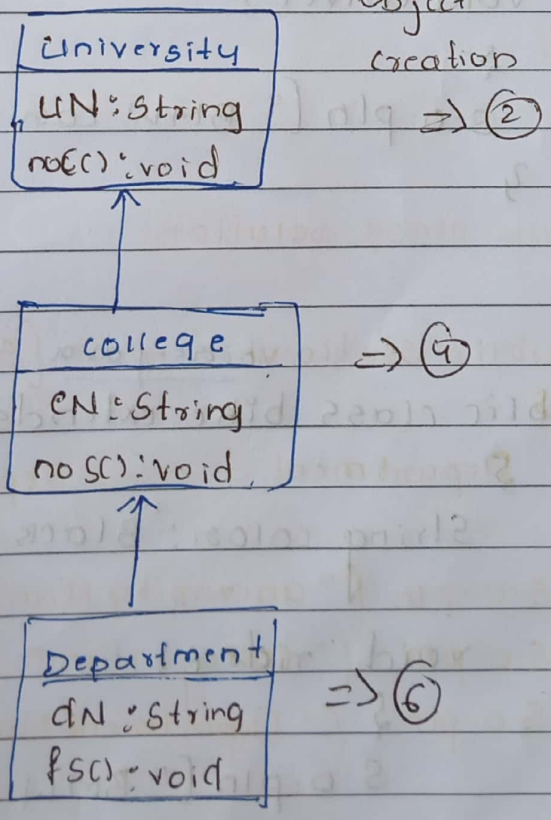
output

University Name : VTU
College Name : Jspiders
Department Name : CSE

- - - - - -

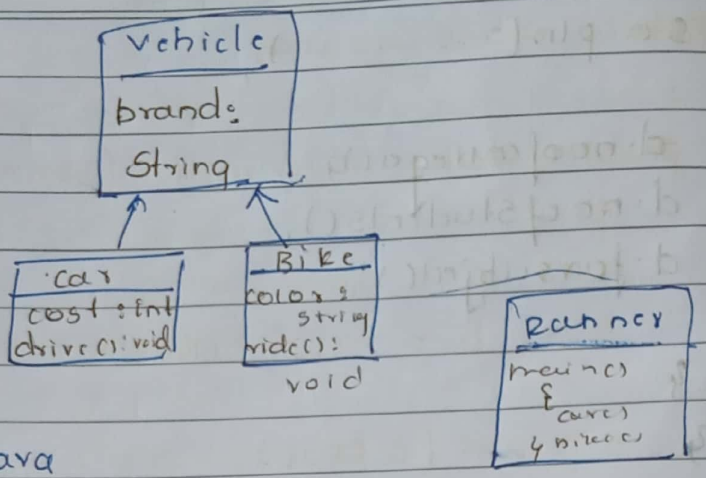
No of colleges are 120
No of students are 400
Fav subject is JAVA

Class diagram



Date: / /

Q)



vehicle.java

```
public class vehicle
  {
    String brand : "BMW";

  }
```

car.java

```
public class car extends vehicle
   {
     int cost : 1000;

    void drive()
     {
     s.o.pln (" BMW car cost is 1000");
     }

}
```

bike.java

```
public class bike extends vehicle
    {
      String color : "Black";

      void ride()
        {
          s.o.pln (" BMW color is black");
    } }
```

Date: / /

runner.java

```
Public class runner
{
public static void main (String[] args)
    {
    car c = new car ();
    Bike b = new bike();

    S.o.pln (c.cost);
    S.o.pln (b.strincolom);
    S.o.pln (" - - - ");
    c.drive();
    b.ride();
    }
}
```

Date: 18/2/2020

Super Keyword

* Super is a Keyword which is used to point to the or access superclass properties
* Super keyword can be used with variables & methods

Syntax:

```
super.variableName
super.methodName();
```

Note: In order to use super keyword, Inheritance is mandatory

Ex

Father.java

```
package com;

public class Father
 {
   int x=10;
 }
```

Son.java

```
package com;
public class son extends Father
 {
    int x=20;

   void display()
   {
     int x=10;
     s.o.pln(x);
     s.o.pln(this.x);
     s.o.pln(super.x);
   }
```

```
public static void main (String[] args)
  {
    Son s = new Son();
      s.display();
  }
}
```

O/P

10

20

30

Method chaining

Method chaining is a process of 1 method calling another method.

Ex

Father.java

```
public class Father
 {
   void work()
    {
     s.o.pln("Father is Working")
    }
}

// this keyword same class methods
public
// super keyword another class method i.e super
// class method

// in order to use super keyword is-a relationship
// mandatory
```

son.java

```
public class son extends Father
{
    void eat()
    {
        this.play();
        super.work();
        S.O.pln("eating");
        play();
    }

    void play()
    {
        S.O.pln("playing");
    }

    pvsM (String[] args)
    {
        Son s = new son();
        s.eat();
    }
}
```

O/P

```
playing
father is working
Eating
playing
```

jdk 1.3

papergrid

Date: / /

Method Overriding

* Method Overriding is a process of inheriting the method & changing the implementation or definition of the inherited method
* The following rules must be followed in order to achieve method overriding

Rules

Rule1 MethodName should be same

Rule2 Arguments should be same

Rule3 Return type should be same

Note: Access specifier should be same or of higher visibility

Note: In order to achieve methodoverriding, Inheritance is necessary

Optionally, While Overriding we can make use of annotation Override, @Override.

* @Override is an indication to the pgmer & JVM that this method is inherited & Overridden
* @Override was introduced from jdk 1.5

Ex 3)

Father.java

```
public class Father {
{
    void bike()
    {
      s.o.pln ("Father's Bike");
    }
}
```

papergrid

Date: / /

Son.java

```
public class son extends Father
{

    @Override
    void bike()
    {
        S.O.pln("Son's Bike");
    }

    void eat()
    {
      S.O.pln("Son is Eating");
    }
}
```

Solution.java

```
public class Solution
{
   PVSM(String[] args)
   {
      Son s = new Son();
       s.bike();
       s.eat();
   }
}
```

Olp

```
Son's Bike
Son is Eating
```

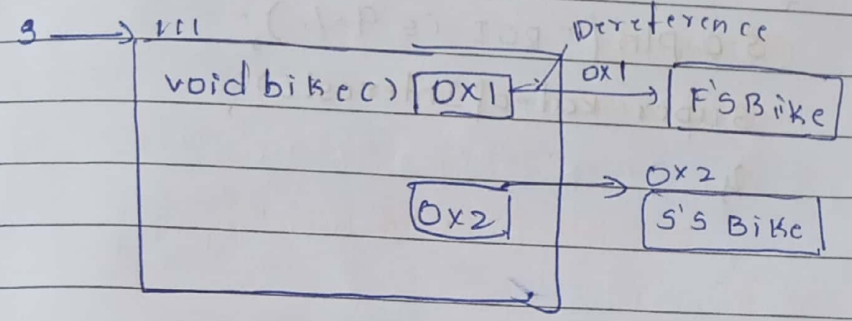
Date: / /

void m1() → Method Declaration / Head — understanding

{

= → Method implementn / Defn / Body

}



Ex Bank.java

```
public class Bank
 {
   void RateofInterest()
     {
      s.o.pln("ROI is 0%");
     }
 }
```

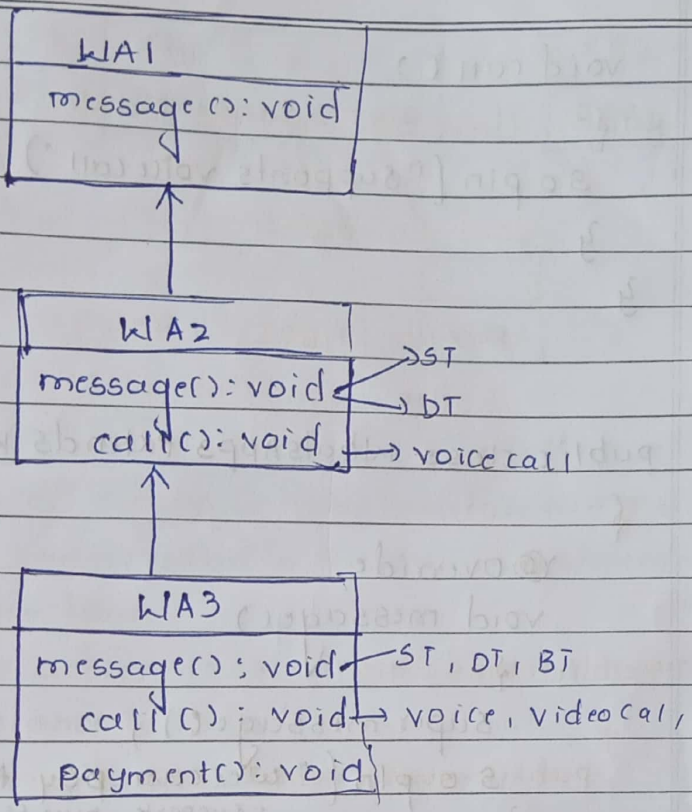
```
public class ICICI extends Bank
 {   @Override
   void RateofInterest()
   {
     Super.RateofInterest();
     s.o.pln("ROI is 6%");
   }
 }
```

```
public class SBI extends Banls
{
    void RateofInterest()
    {
      S.o.pln("ROI is 9%.");
      super.RateofInterest();
    }
}

public class test
{
  public static void main(String[] args)
    {
      ICICI i = new ICICI()
      i.rateofInterest();

      S.o.pln("====");

      SBI s = new SBI()
        s.rateofInterest();
    }
}
```

o/p

```
ROI is 0%.
ROI is 6%.
====
ROI is 9%.
ROI is 0%.
```

Date: / /



1) post images } single
2) GIFS } single
3) videos } 3

Q) public class WhatsApp1
{
void message()
{
S.o.pln ("supports singleTicks");
}
}

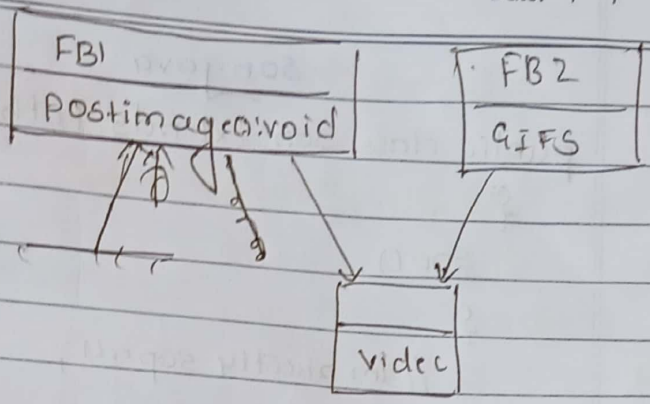
public class WhatsApp2 extends WhatsApp1
{
@Override
void message()
{ → super.message();
S.o.pln ("supports Doubleticks");
}
}

```
void call()
{
  s.o.pln("supports voice call");
}
}
```

```
public class WhatsApp3 extends WhatsApp2
{
  @Override
  void message()
  {
    super.message();
    s.o.pln("we can pay through WhatsApp");
  }
  @Override
  void call()
  {
    s/o/p super.call();
    s.o.pln("supports video call");
  }
}
```

```
public user
{
  psvm(String[] args)
  {
    WhatsApp3 a = new WhatsApp3();
    a.message();
    a.call();
    a.payment();
  }
}
```

Date: / /



Constructor chaining w.r.t Inheritance

* The process of subclass constructor calling superclass constructor is constructor chaining w.r.t Inheritance.
* In order to achieve constructor chaining in another class inheritance is mandatory
* We make use of super calling statements
* In order to achieve constructor chaining in another class.
* Constructor chaining are is divided into 2 ways,
  (i) implict constructor chaining
  (ii) Explict constructor chaining

Implicit Constructor chaining

* When we create an object of a class, if that class has a superclass & if that superclass has a non parameterized constructor that gets executed implicitly

Father.java

```
public class Father
  {
      Father()
      {
       s.o.pln (1);
  }
}
```

Son.java

```
public class Son extends Father
{
    Son()
    {
        // implicitly super();
        S.O.pln(2);
    }
}
```

Test.java

```
public class Test
{
    public static void main (String[] args)
    {
        S.O.pln("start")
        Son s = new Son();
        S.O.pln("end");
    }
}
```

Output

```
start
1
2
end
```

Explicitly constructor chaining

* When we create an object of a class, if that class has a superclass, & that superclass has a parameterized constructor, then subclass constructor has to invoke superclass constructor Explicitly else we get compile time error

Father.java

```
public class Father
  {
    Father(int x)
    {
    S.O.pln(1);
    }
}
```

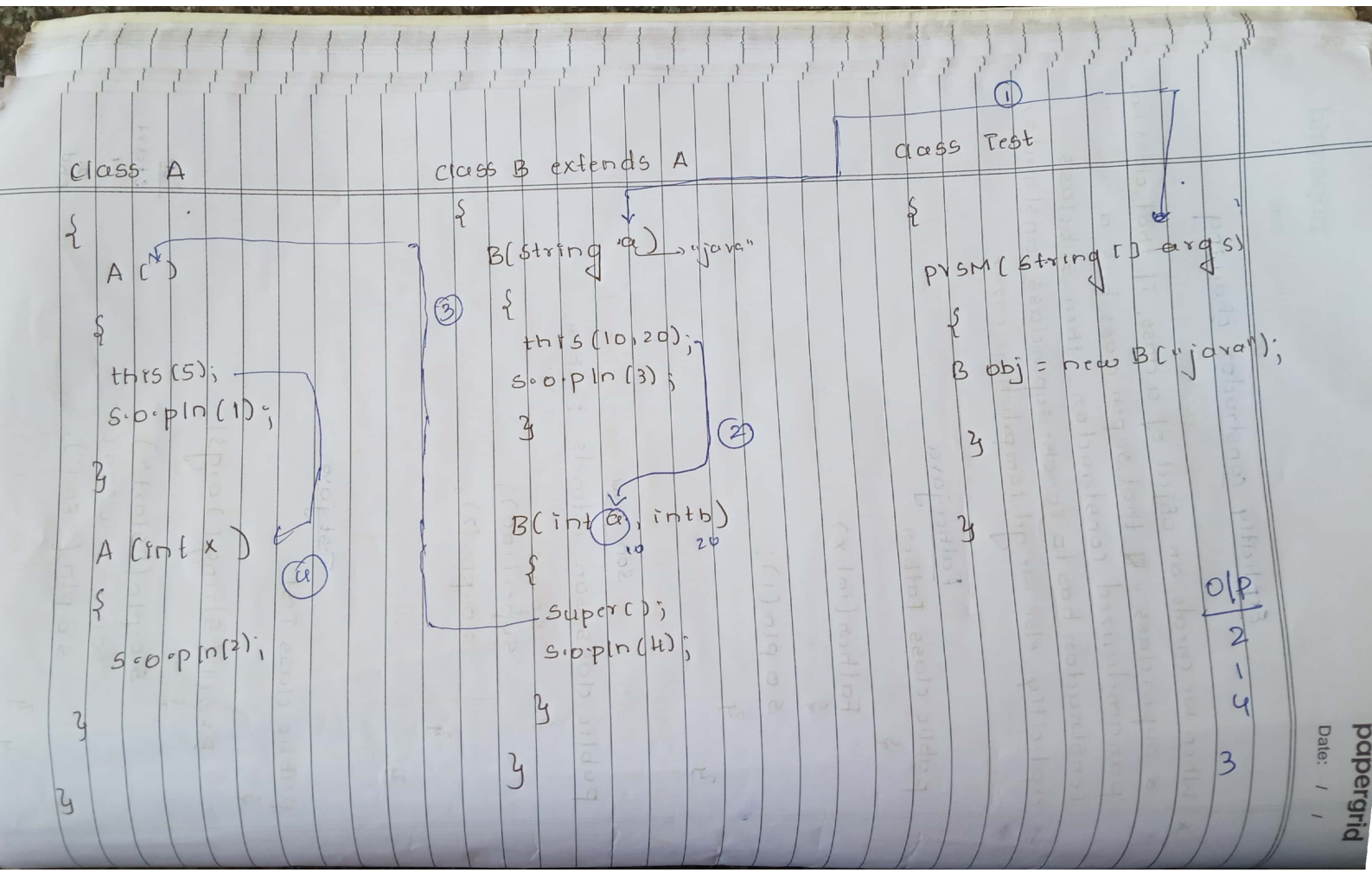
Son.java

```
public class son extends Father
    {     son()
          {
          super(100);
          S.O.pln(2);
    }
}
```

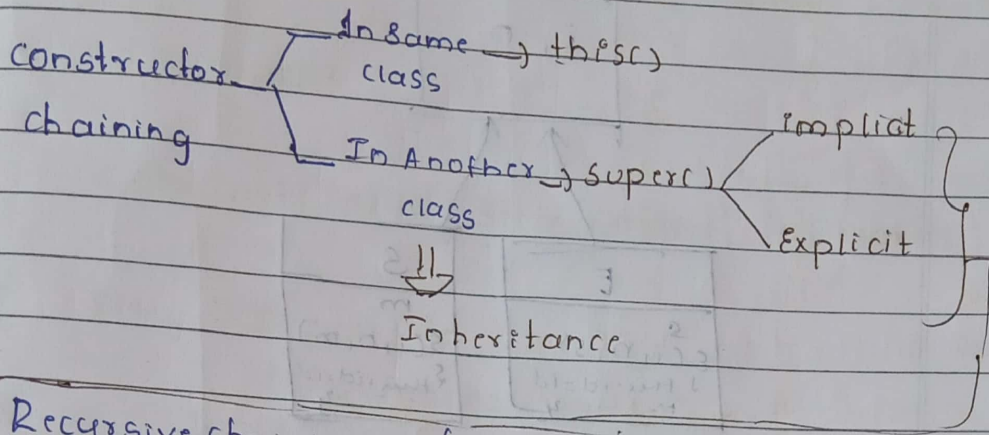
Test.java

```
public class Test
{
   pvsM(String[] args)
     {
       S.O.pln("start")
       Son s = newson();
       S.O.pln("End");
}  }
```

o/p
start
1
2
End



Date: / /

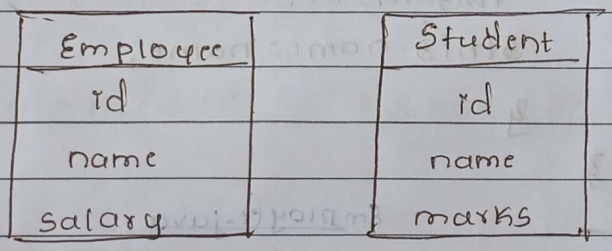


1. Recursive chaining X (Not possible in java)
2. 'n' constructor ⇒ n-1 calling statements
3. First executable stmt → this() / super()
4. super() & this() cannot be present in same Constructor

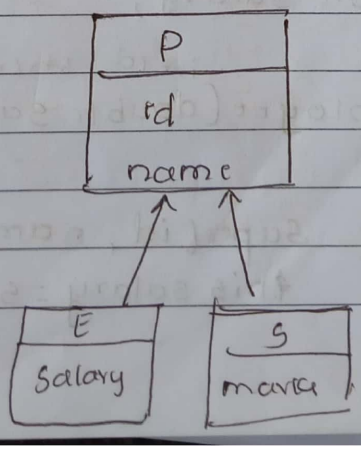
Advantages of Inheritance

- code Reusability is increased
- code Redundancy / Repetation is avoided

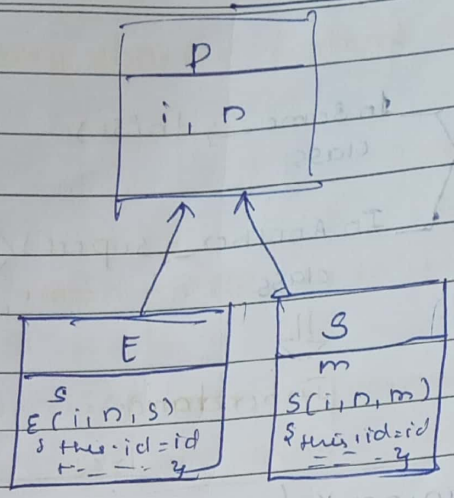
pblm statement



solution



Pblm Stmt



Soln

Person.java

```
public class person
{
   int id;
   string name;
   person(int id, string name)
      {
         this.id=id;
         $this.name=name;
      }
   }
```

Employee.java

```
public class Employee extends person
  {
   double salary;
                    int id, string name,
   Employee(double salary)
     {
        super(id, name);
        this.salary=salary;
  }}
```

Date: / /

Student.java

```
Public class student extends person
{
    int marks

    student(int id, string name, int marks)
    {
Err.    super(id, name);
        this.marks = marks;
    }
}
```

Runner.java

```
public class Runner
{
    prsM (string[] args)
    {
    Employee e = new employee (1, "D", 124);

s.o.pln("e.id+" "+e.name+" "+e.salary);

    student s = new student (2, "A", 12);

s.o.p ln(s.id+" "+s.name+" "+s.marks);

    }
}
```

output

```
1 D 124
2 A 12
```

20/2/2020

Access Specifiers

* Access specifier is used to specify the accessibility of a specific member
* In other words, it is used to indicate the boundary of a member
* Different access specifier are as follows

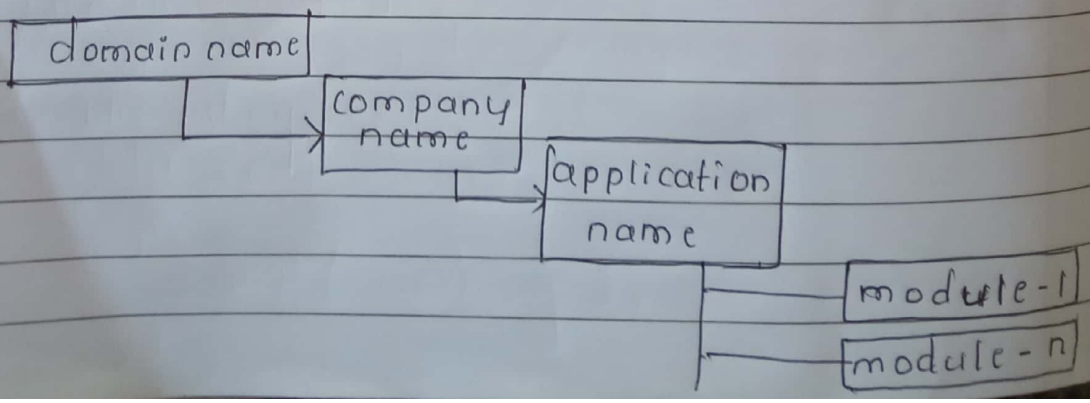
1) public
2) protected
3) Default
4) private

	Outer Class	Inner/Nested Class	Variable	Method	Construct
public	✓	✓	✓	✓	✓
private	X	✓	✓	✓	✓
default	✓	✓	✓	✓	✓
protected	X	✓	✓	✓	✓

Packages :-

* packages are folder or directory
* Maintainance becomes easy when we create packages
* Searching also becomes easy

Structure of a package

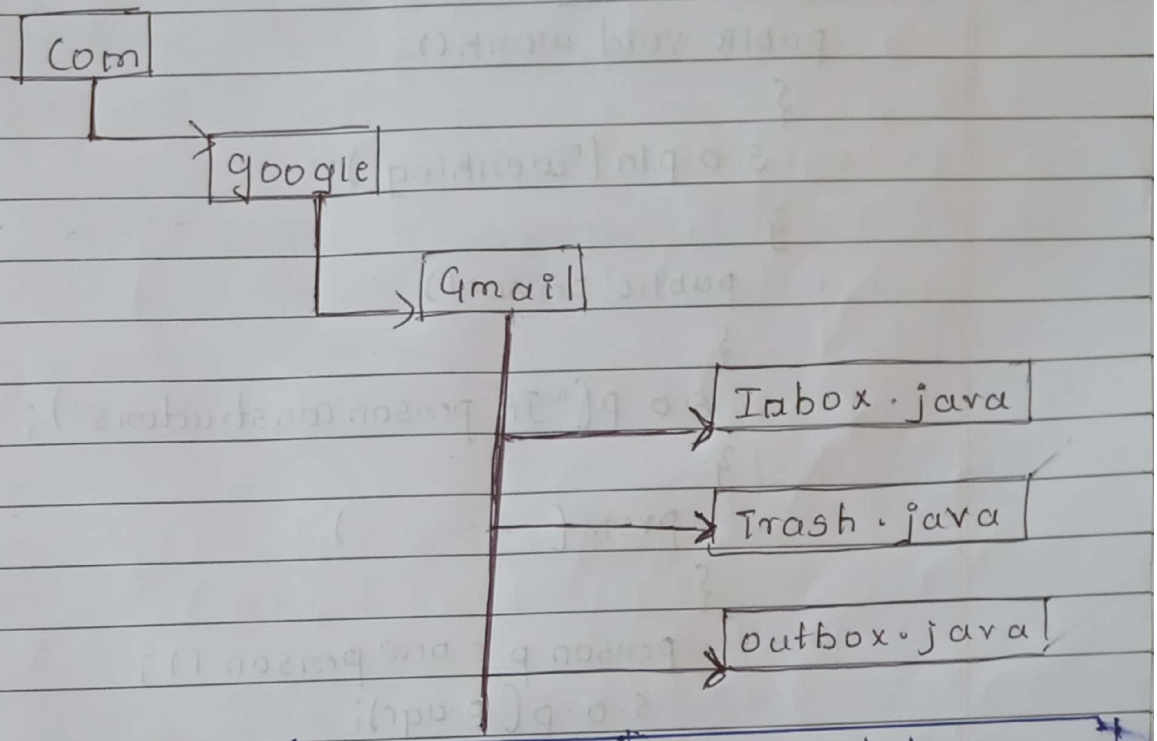


Ex:

* Example

com.google.gmail

Gmail Application



	In same class	Different class Same package	Different class different package
public	✓	✓	✓
protected	✓	✓	✓
default	✓	✓	X
private	✓	X	X

import
Is-A relationship

NOTE: public is an access specifier where in it can be accessed globally

→ Accessing Public Properties in same class.

Ex:-
```
package com.jspiders;
public class person
{
    public int age=10;
    public void work()
    {
        s.o.pln("working");
    }
    public person()
    {
        s.o.p("In person constructors");
    }
    psvm(---)
    {
        person p = new person();
        s.o.p(p.age);
        p.work();
    }
}
```

O/P :- In person constructor
10
working

Note:-
→ Accessing public properties in another class, same package

```
package com;
public class Employee
{
    public int age=30;
    public void work()
    {
        s.o.p("working");
    }
```

Accessing public properties in another class Same Pack

Ex
```
package com;
public class Employee
{
 public int age=30;
}
```

O/P:- 30

```
package org;
import com.Employee;
public class Test
{
  psvm(---)
  {
    Employee Emp = new Employee();
      s.o.p(employee emp.age);
  }
}
```

Importing all the classes from a specific package

```
public class Demo
{
package com.jspiders.example;
  public class Runner
  {
  }
```

```
package org;
  import com.jspiders.example.*;
// import com.jspiders.example.Demo;
// import com.jspiders.example.Runner;
      pubic class Test
    {
      psvm(---)
```

```
{
    Demo d = new Demo();
    Runner r = new Runner();
}
```

Private

Private is an access specifier where in private properties can be accessed only within the same class

Ex:-

```
package com.jspiders.example
    public class Demo
    {  ——→ private static int x;
       psvm (---)
       {
          s.o.p(x); // Demo.x
       }
    }
```

```
package com.jspidens.example
    public class solution
       {
          psvm (---)
          {
             s.o.p(Demo.x) // Demo.x
                           error
          }
    }
```

21/2/2020

Default:

* Default is an access specifier where in it can be access within the same class same package
* When we don't use any of the access specifier then it is a automatically default.
* Default means package level

Ex: //Accessing default properties in same class

```
package com;
class Mobile
{
 int cost=2000;
  psvm (---)
   {
    Mobile m = new Mobile();
       s.o.p (m.cost);
   }
}
```

O/P : 200

Ex: // Accessing Default properties in another class but same package

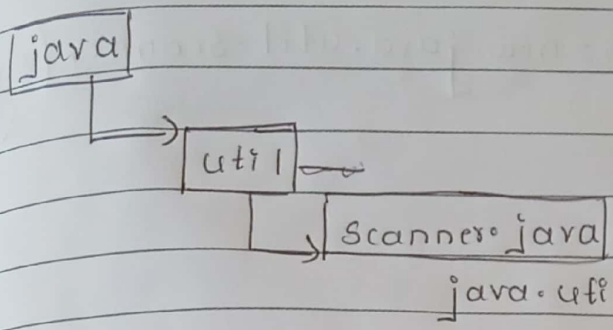
```
package com;
class student
{
 static int rollNO=123;
    void study()
      {
       s.o.p ("studying");
      }
}
```

```
package com;
public class solution
  {
     PrSM(- - - -)
     {
       S.O.P(student.rollNO);
       Student s = new student();
             s.study();
      }
}
```

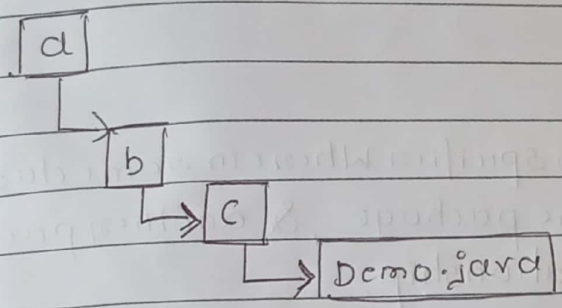
o/p :- 123
studying

Fully Qualified class Name.

* Fully qualified class name is considering the class name from its package level



java.util.Scanner.java



a.b.c.Demo.java

Ex)

```
package com;
public class Pen
  {
    public static int cost=20;
     public int x=10;
  }

package org;
 //import com.pen;

//import java.util.Scanner;
Public class Runner
  {
    psvm(---)
    {
    s.o.p(com.pen.cost);
  {
```

```
com.pen p= new com.pen();
    s.o.p(p.x);

java.util.Scanner s= new java.util.Scanner(system.in);
  }
}
```

O/P : 20
10

Protected

* protected is an access specifier where in same class, same package & outside the package, & another package incase of Is-A-relationship.

⇒ Accessing protected members outside the package

```
package com;
public class Father
{
  protected int age=40;
}
```

```
package org;
import com.Father;

public class son extends Father
  {
   psvm (----)
    {
      son s = new son();
       s.o.p(s.age);
    }
}
```

O/P : 40

Final keyword

* Final is a keyword which can be used with a variable, method, class
* Final variables cannot reinitialized i.e it acts as constant
* Final method can be inherited but cannot be Overridden.
* Final class cannot be inherited at all

=> Coding standards / Naming Conventions By Oracle

(1) Class => Car, AudiCar, SortStudentsById

(2) variable => age, employeeSalary, noOfStudents

(3) method => test(), checkEvenOrOdd()

(4) Constants => PI, EXP

(5) packages => lowercase com, org ____

Encapsulation

* Encapsulation is a process of binding & protecting data members & member function in a single entity called class
* Binding can also be referred as Grouping / Wrapping
* Data members are also referred as states / Variables
* Member Functions are usually referred as the Behaviour / Methods
* Best example for encapsulation is JavaBean class

Specifications for creating Java Bean class

1) Create a public non abstract class
2) Declare private Data Members
3) Define public setter method & getter method

Not imp

4) Class has to implement, serializable interface
5) class should have a public non parameterized constructor

Problem Statement

```
public class person
{
    private int age;
}
```

```
class Test
{
    psvm (String[] args)
    {
        person p = new person();
        S.o.p(p.age);
        p.age = 60;
    }
}
```

age cannot be accessed as it's private

(We cannot print/ modify the value)

- Setter method sets the value
- Getter method returns the value.

```
public class person
{
private int age;
public void setAge(int age)
{
    this.age=age;
}

public int getAge()
{
return this.age;
// age;
}
}
```

```
class Test
{
PVSM (String[] args)
{
person p = new person();
p.setAge(25);
int age = p.getAge();
S.O.P(age);
// or
S.O.P(p.getAge());
}
}
```

Q) i) Create a public non abstract class called as employee
ii) Declare to private variables as id, name.
iii) Have public getter & setter method respectively
iv) create another class called as solution & under main method invoke those respective method

⇒

```
public class employee
{
private int id;
private string name;

public void setId(int id)
{
    this.id = id;
}
```

```
public int getId();
  {
return this.id=id;
        id.
  }

public void setName (string Name)
{
   this.name=name;
}

public string getName()
 {
    return name;
 }
}
```

solution.java

```
Class solution
{
 psvM(- - - )
{employee e = new employee()
   p.setId(100);
   p.setName(Bhagya);
System.out.println(p.getId());
System.out.println(p.getName());
   }
}
```

O/P

100

Name

// logical failed

3/2/2020

Advantages of encapsulation on Java Bean class,

1) Variables can be made read only by having only getter method & the data can be made write only by having only setter method
2) The data can be made read & write by having both the setter & getter method.

8)

Employee.java

```
public class Employee
  {
    private int id;
    private String designation;
    private double salary;

public Employee (int id, string designation, double salary)
   {
     this.id=id;
     this.designation = designation;
     this.salary = salary;
  }

public int getId()
  {
    return id;
  }

public void setSalary(double salary)
  {
       this.salary = salary;
        S.O.pln ("salary got updated");
  }
```

```
public void setDesignation (String designation)
 {
     this.designation = designation;
 }

public String getDesignation()
 {
     return designation;
 }
}
```

// Since we have only getId() we can only get the id but cannot modify the id
// since we have only setSalary() we can only set the salary but cannot get the salary
// we can get & set designation i.e read and write both

Test.java

```
public class Test
{
  psvm (String[] args)
{
  Employee e = new Employee (101, "Developer", 200.34);
    s.o.p("Id :" + e.getId());

  e.setSalary(500.56);

  s.o.pln(e.getDesignation());

  e.setDesignation ("project Manager");
     s.o.pln(e.getDesignation());
}
}
```

O/P

- Id:101
- Salary got updated
- Developer
- project Manager

2) We can protect the data from illogical initialization

a) Person.java

```
public class person
{
  private int age
  private string email;

  public void setEmail (String email)
  {
     if(email.contains("@"))
     {
        this.email=email;
        S.o.pln ("Email is updated");
     }

  else
  {
     s.o.pln("Invalid email");
  }
}

public string getEmail()
  {
     return email;
  }

  public void setAge (int age)
  {
     if(age>0)
     {
```

```
        this.age = age;
        S.O.println("Age is valid");
        }

    else
        {
        S.O.println("Age is invalid");
        }
}

public int getAge()
    {
        return age;
    }
}
```

Testperson.java

```
public class testperson
    {
    P VSM (string[] args)
    {
    person p = new person();

        p.setAge(-30);
        S.O.println(p.getAge());

        p.setAge(30);
        S.O.println(p.getAge());

    p.setemail("xyz.com")
        S.O.println(p.getemail());
```

```
p.setEmail("tom@gmail.com");
    S.o.pln(p.getEmail());
  }
}
```

O/P

2)
```
public class usco
{
 private long mno;

public void setMobileNO (long mno)
  {
    if (length==10)
    {
        this.mno=mno;
        S.o.pln("MobileNo is invalid");
    }
   else
    {
      S.o.pln("MobileNo is Invalid");
    }
  }

public long getMobileNO ()
  {
    return mno;
  }

}
```

Solution.java

```
import java.util.Scanner;
class solution
{
  psvm (String[] args)
  {
    Scanner s = new Scanner (System.in);
    s.o.pln("Enter a mNo");
    long no = s.nextLong();
```

4/2/2020

Abstract

Abstract is a keyword which can be used with class & Method

Concrete class : A class which is not declared using abstract keyword is called concrete class

* Concrete class can allow only concrete methods

abstract class, methods

Ex :- Concrete class / Non Abstract class

```
class A
{
  // concrete Methods
}
```

Abstract class

A class which is declared using abstract keyword is called as Abstract class

* Abstract class can allow both Abstract methods & Concrete methods

Ex :-

```
abstract class B
{
// Both abstract & concrete methods
}
```

④ Concrete Method: A method which has both declaration & implementation is called as concrete method

Ex: void test() → Method Declaration (Head)

{
- - - -
}

→ Method implementation / Definition

⑤ Abstract Method: A method which has only declaration & no implementation is referred as Abstract method

* Abstract method has to be declared using abstract keyword

Syntax: Accessspecifier abstract returntype methodName(); (Arguments)

Ex:- 1) public abstract void display();

2) _____ abstract void test();
default.

Rules which has to be followed when a class inherits an abstract class (OR) contract of abstract

RULE: 1) When a class inherits an abstract class, it is mandatory to override all abstract methods

2)

Example for Rule 1

Ex

```
abstract class person
{
 abstract void eat();
}
```

```
class student extends person
{
    @Override
    void eat ()
    {
    s.o.p ("eating");
    }

    psvm (- - - )
    {
    student s = new student()
      s.eat();
    }
}
```

Rule 2: If the abstract method cannot be overridden, make the subclass as abstract.

ProjectManager.java

```
public abstract class projectManager
{
   abstract void giveTask();
}
```

Developer.java

```
public abstract class Developer extends projectManager
{
   abstract void developAp();

   void display()
    {
      s.o.pln("HELLO");
    }
}
```

Intern.java

```
public class Intern extends Developers
{
  @Override
  void giveTask()
  {
    S.O.pln("study java");
  }

  @Override
  void developApp()
  {
      S.O.pln("Develop some" + "mini project);
  }
}
```

Solution.java

```
pubic class solution
{
pvsM (String [] args)
{
  Intern i= new Intern();
     i.developApp();
     i.giveTask();
     i.display();
  }
}
```

O/P

-Develop some mini project
study java
Hello

NOTE : We cannot create an object of abstract class (bcz it is an incomplete class)

2) Abstract Methods cannot be private

3) Abstract Methods cannot be static → bcz

4) Abstract Methods cannot be Final

(*) Abstract class can have constructors & those constructor, public are invoked either implicitly or explicitly using Super calling statement

(*) Subclass should be a concrete class

Father.java

```
public abstract class Father
{
    Father(int x)
    {
      System.out.println(1);
    }
}
```

Son.java

```
public class son extends Father
{
   son()
   {
     super(10);
     system.out.println(2);
   }

}
```

Test.java

```
public class Test
{
  psvm(String[] args)
  {
    Son s = new Son();
  }
}
```

O/P

1

2

5/3/2020

Interface

Interface is a JavaType Definition which has to be declared using interface keyword

- Interface can also be referred as a medium b/n 2 s/m's
- wherein, 1 s/m will behave as client & another system will behave as system with resource

Syntax : interface InterfaceName

```
{

}
```

- It is possible to have a variable in an interface & those variables are automatically public, static, Final
- Interface can allow only abstract methods & those methods are automatically public & abstract

```
interface Test
{
  int x = 50;      // public static Final
  void display();  // public abstract
}
```

* Interface doesnot contain any constructor therefore We cannot create an object of interface.

* A class can achieve Is-a-relationship (Inheritance) with an interface using implements keyword.

*Note: When a class implements an interface, it is mandatory to Override abstract method

*Note: The Access Specifier should be same or of Higher visibility

```
interface College
{
  public void enjoy();
}

Class Student implements College
{          @Override
  public void enjoy()
  {
    s.o.p("Enjoyed");
  }

psvm(____)
{
  Student s = new student();
    s.enjoy();
}
}
```

same

Is-A	Class	Interface
Class	extends	implements
Interface	X	extends

Father.java

```
package org;
public interface Father
{
  int age=40; //public, static, final i.e public static
                final int age=40;
  void work(); // public and abstract
}                i.e public abstract void work();
```

Son.java

```
package org;
public interface Son extends Father
{
   void enjoy();
}
```

Grandson.java

```
package org;
public class Grandson implements Son
{
   @Override
     public void work()
     {
      S.O.Pln ("working");
     }

   @Override
     public void enjoy()
     {
} }     S.O.Pln ("enjoying");
```

Runner.java

```
package org;

public class Runner
{
 public static void main (String[] args)
  {
    S.o.pln( Father.age);
   Grandson gs = new Grandson();

    gs.work();
    gs.enjoy();
   }
}
```

o/p

```
40
Working
Enjoying
```

* A class can implement any number of interfaces
* A class can inherit extend another class & implement any number of interfaces

A.java

```
package org;

public interface A
{
   void m1();
}
```

B.java

```
package org;

public interface B
{
   void m2();
}
```

Person.java

```
package org;

public class person
{
    void eat()
    {
     s.o.pln("Guldu is eating");
    }
}
```

Solution.java

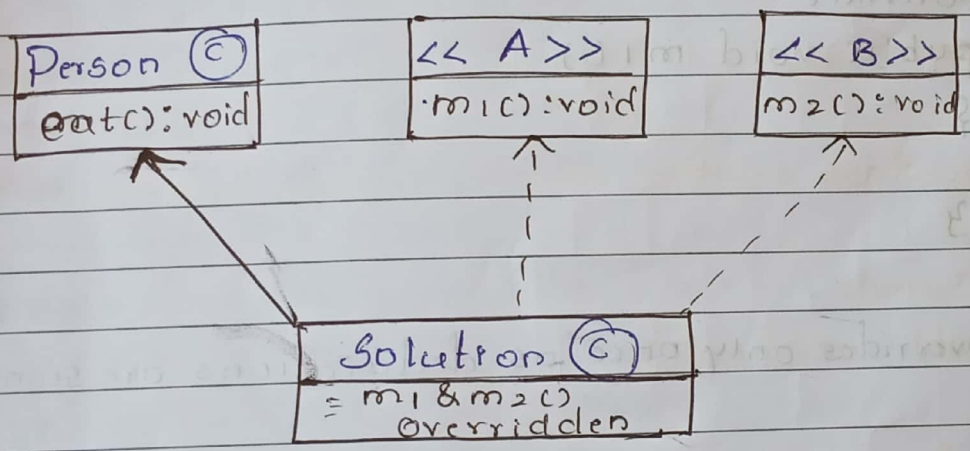
```
package org;

public class solution extends person implements A,B
{
  @Override
    public void m1()
    {
      s.o.pln(" in m1");
    }

@Override
    public void m2()
    {
      s.o.pln("in m2");
    }
}
```

output

```
in m1
in m2
Guldu is Eating
```

Mainclass.java

```
public class MainClass
{
  psvm (String[] args)
  {
    Solution s = new solution();
      s.m1();
      s.m2();
      s.eat();
  }
}
```



<< * >> => Stereotypes used to indicate Interface

Ⓒ => Class

↑ => extends

⇡ => implements

8) A.java

```
package org;

public interface A
{
  void m1();
}
```

B.java

```
package org;

public interface B
{
  void m1();
}
```

Solution.java

```
public class solution implements A,B
{
  @Override
  public void m1()
  {

  }
}
// Overrides only once as declarations are same
```

9) public class

A.java

```
public interface A
{
  void m1();
}
```

B.java

```
public interface B
{
  void m1();
}
```

```
public class solution implements A,B
{
  @Override
  public void m1()
  {
}
```

solution.java

```
public void m1(int a)
{

}
}
// override both as there is change in declaration
```

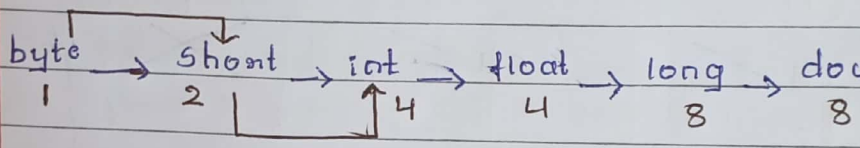
	Instantiation or object creation	Construct-ors	Abstract Methods	Concrete Methods
Non-Abstract class / concrete class	✓	✓	✗	✓
Abstract class	✗	✓	✓	✓
Interface	✗	✗	✓	✓

Type Casting

* Type casting is a process of converting 1 type of data into another type of data
* Type casting / Type Conversion is divided into 2 types
  (i) Widening
  (ii) Narrowing

i) Widening

* Widening is a process of converting smaller type of data into a bigger type of data
* We also refer Widening as implict / automatic casting



Ex)

```
short a=5;        a | 5 |
float b=a;        b | 5.0 |
```

Ex)

```
int x=10;         x | 10 |
double y=x*10;    y | 100 |
```

ii) Narrowing

* Narrowing is a process of converting bigger type of data into smaller type of data
* Narrowing is also referred as Explicit Typecasting

Ex)

```
int i=10;              i | 10 |
byte j=(byte)10;       j | 10 |

double a=5.75;         a | 5.75 |
int b=int(a);          b | 5 |
```

Ex>

```
public class casting
{
psvm(String[] args)
{
  s.o.pln("Implicit Casting");

   int a=10;
double b=a;
  s.o.pln(a+" "+b);

  char c='a';
   int d=c;
  s.o.pln("----");
  s.o.pln("Explicit Casting");

  double e=5.67;
   int f=(int)e;
   s.o.pln(e+" "+f);

  int x=66;
  char y=(char)x;
  s.o.pln(x+" "+y);

  s.o.pln("------");

  s.o.pln((char)65);

  s.o.pln("----");

  s.o.pln("a"+10);
  s.o.pln('a'+20);
  s.o.pln('A'+'B');
```

output

Implicit Casting
10 10.0
a 97

Explicit Casting
5.67 5
66 B

A

a10
117
131

Interview

ASCII : (American Standard Code Information Interchange)

Interview (Imp**)

Q) Write a program to print alphabets from a to z & their respective Ascii values

⇒
```
class Ascii
{
 psvm (String[] args)
  {
     for (int i=65 ; i<=96 ; i++)
      {
        S.o.pln(" The Ascii value of " +(char)i+ "is"+i);
      }
  }
}
```

```
    for (char i='A' ; i<='z' ; i++)
      {
        S.o.pln (int(i));
      }
  }
}
```

✱ Interview (Imp***)

Q) Why do we specify float the last when we initialize float variable.

⇒ In java, whenever the compiler comes across a decimal numeric value it assumes as double i.e 8 bytes

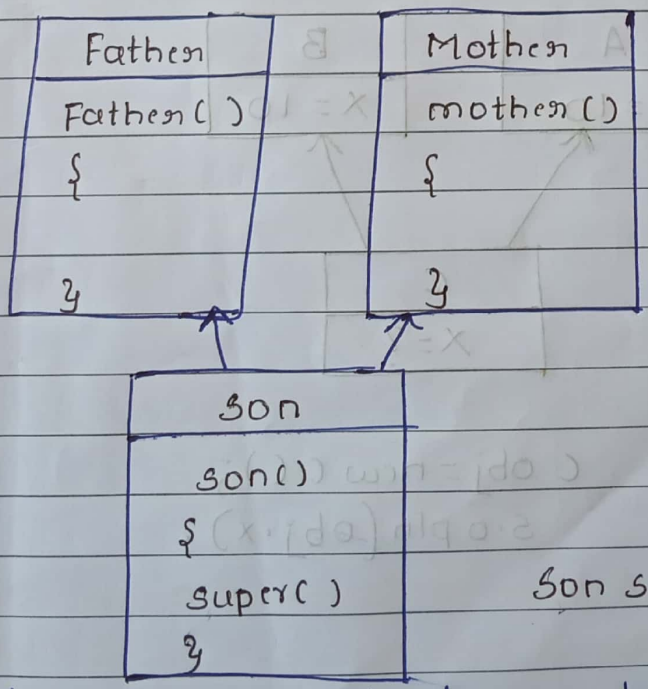
∴ When we try to store it in float, we are casting it to 4 bytes i.e Converting double to float.

Imp** public class Solution

```
{
pvsm(String[] args)
 {
   byte a=5;
   short b=55;
   int  c=555;
   long mobNo=9739070632; // long mobNo=9739070632L
                          // or 9739070632l
   s.o.pln("====");

   float x=6.7f; // float x=6.7F or float x=(float)6.7;
   double y=5.5;
 }
}
```

** Q) Why java doesnot support multiple Inheritance?

→ Ambiguity during Constructor chaining



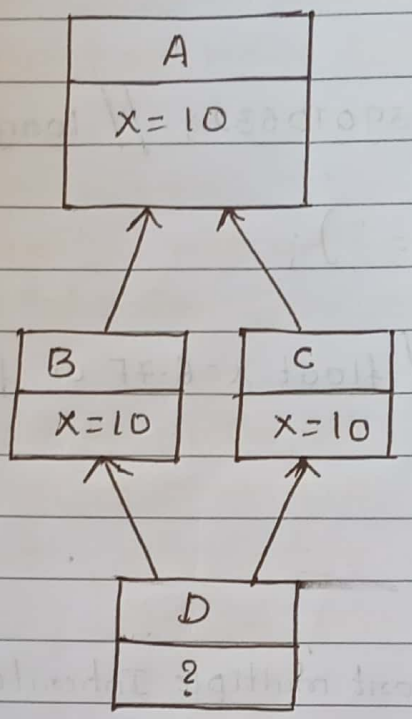
It doesnot understand which superclass constructor to be invoked i.e confusion / ambiguity

2) Diamond problem

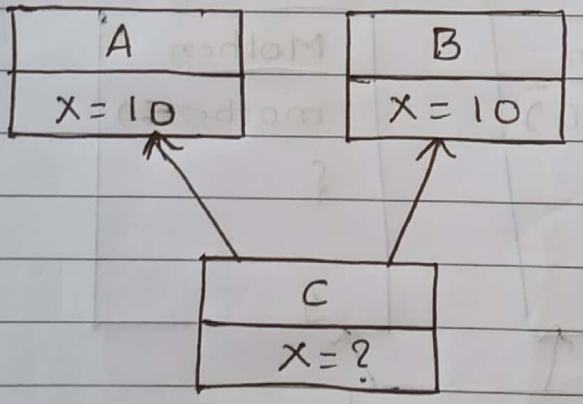
There is a problem created / Ambiguity Created when Superclasses has same variables on methods

∴ Java doesnot support multiple Inheritance

(*)



(*)




```
C obj = new C();
S.o.pln(obj.x);
```

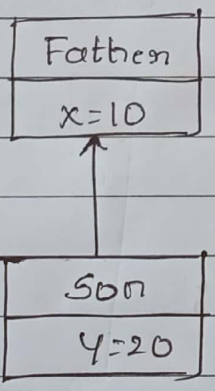
Upcasting & Downcasting

* Upcasting is a process of creating an object of subclass & storing its address into reference of type superclass,
* In order to achieve upcasting Is-A relationship is mandate i.e inheritance
* Upcasting always happens implicitly
* With the upcasted reference we can access only super class properties but not subclass properties

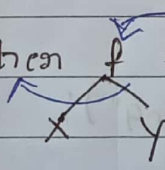
Downcasting

* The process of converting the upcasted reference back to subclass type reference is called as Downcasting
* Downcasting has to be done explicitly
* Downcasting can be achieved only after upcasting
* With the downcasted reference / subclass type reference we can access whole superclass & subclass properties

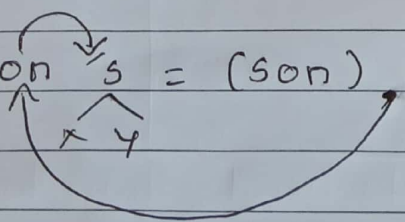
1>



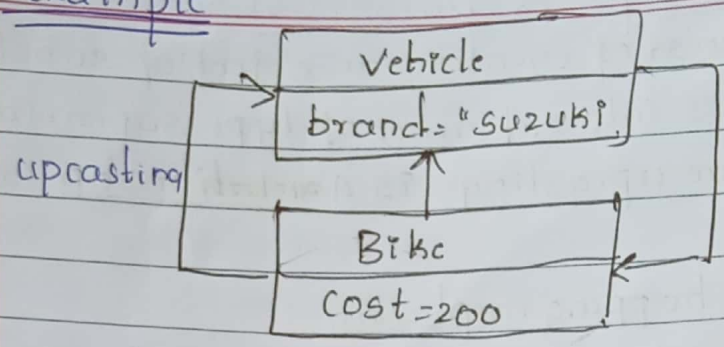
(*) Father f = new Son();



(*) Son s = (Son) f;



Example



① upcasting

```
Vehicle v = new Bike();
```

brand cost

```
Bike b = new Bike();
Vehicle v = b;
```

②

```
Bike b = (Bike) v;
```

brand cost

Ex> Developer.java

```
package classtypecasting;

public class Developer
{
  String domain = "Java";

  void work()
  {
    S.o.pln("Developer is working");
  }
}
```

JrEngineer.java

```
package classtypecasting;

public class JrEngineer extends Developer
{
    String name = "Tom";

    void task()
    {
        S.o.pln ("Learn Java");
    }
}
```

Solution.java

```
package classtypecasting;

public class Solution
{
psvm (String[] args)
{
    Developer d = new JrEngineer(); // up-CASTING.
    /* JrEngineer je = new JrEngineer();
       Developer d = je; */

    S.o.pln (d.domain);
    d.work();
    /* S.o.pln (d.name); Error
       d.task(); */

S.o.pln ("= = =");
```

```
JrEngineer jr = (JrEngineer) d;
S.o.pln (jr.domain+" "+jr.name);
   jr.work();
   jr.task();
  }
}
```

output

```
Java
Developer is working
= = = = = =
Java Tom
Developer is working
Learn java
```

10/3/2020 Interview (Imp)

* Static methods can be Overloaded but cannot be Overridden
* Non static methods can be both Overloaded & Overridden as well

Note: If a method has been overridden & if we invoke a method with the upcasted reference or downcasted reference always a Overridden implementation will get executed.

Q)

Father.java

```
package classtypecasting;

public class Father
  {
    void bike()
    {
      S.o.pln("old fashioned bike");
  } }
```

Son.java

```
public class son extends Father
{
  @Override
   void bike()
   {
    s.o.pln ("Modified Bike");
   }
}
```

Test.java

```
Public class Test
  {
   prsM (string [] args)
   {
     Father f = new son();
      f.bike();

     son s = (son) f;
       s.bike();
    }
}
```

Output

New modified bike
New modified bike

8) Why do we get ClassCastException ?

→ If 1 class type has been upcasted, we have to downcast to the same class type otherwise we get ClassCastException i.e if 1 class type has been upcasted & downcast to some other class type

* We also get ClassCastException when we downcast without upcasting .: In order to avoid it we make use of instance of operator

Instanceof Operator

Instanceof is an operator which is used to checks if an object is an instance of specific class or not. In other words, Instanceof is used to checks if an object is having the properties of specific class or not. Instanceof always returns boolean values.

Syntax: object instanceof ClassName

Ex)

Father.java

```
package classtypecasting;          public class
 public class Father
 {
    String = "Tom";
}
```

Son.java

```
public class Son extends Father
{
   int age = 25;
}
```

Daughter.java

```
public class Daughter extends Father
 {
    String hobbies = "Tik Tok, Dancing";
}
```

Test.java

```
public class Test
{
  psvm (String[] args)
  {
    s.o.pln (new son () instanceof son);
    s.o.pln (new son () instanceof Father);

    s.o.pln ("----");

    Son s = new Son();
      s.o.pln (s instanceof son);
      s.o.pln (s instanceof Father);

    s.o.pln("---");

    Daughter d = new Daughter();
      s.o.pln (d instanceof Father);
      s.o.pln (d instanceof Daughter);

    s.o.pln("----");

    Father f = new Father();
    s.o.pln (f instanceof Father);
    s.o.pln (f instanceof son);
    s.o.pln (f instanceof Daughter);
  }
}
```

O/P

```
true
true
----
true
true
----
true
true
false
false
```

Mainclass.java

```
public class MainClass
{
  PVSM (String[] args)
  {
    Father obj = new son();

   if (obj instanceof son)
    {
     S.O.pln("Downcasting to son").
     Son s =(son) obj;
     S.O.pln (s.name+" "+s.age);
    }

   elsif (obj instanceof Daughter)
   {
    S.O.pln("Downcasting to Daughter");
    Daughter d= (Daughter) obj;
    S.O.pln(d.name+" "+d.hobbies);
   }
}
}
```

O/P

Downcasting to son
Tom 25

Hackerrank

1) Write a java pgm in order to count the no of objects created for a specific class

```
public class Test
{
    Test ()
    {
        count++;
    }
}
```

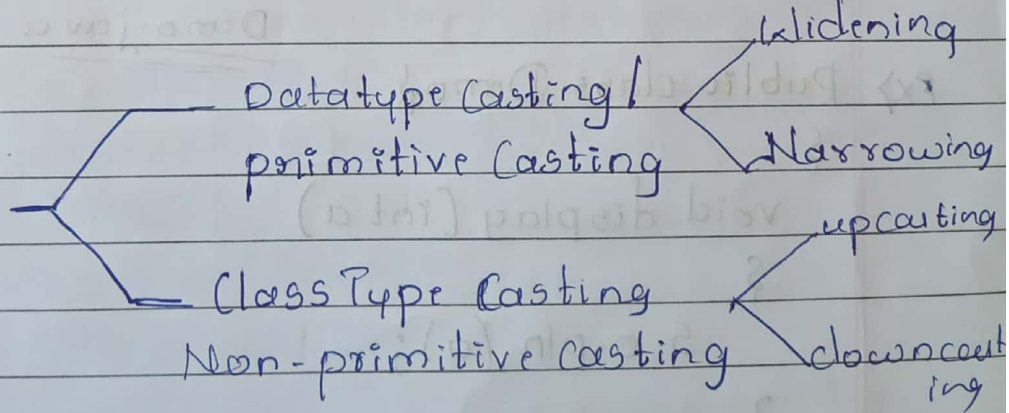
```
public class Test
{
    int count=0;
    Test ()
    {
        S.o.pln("No of times object created" +count);
        count++;
    }

    psvm (String[] args)
    {
        new car()
        car c = new car();
        S.o.pln("No of objects created" +count);
    }
}
```

HR Syed Aqeeb / vignesh

PNO:- 7760866966

PNO:- 7022889639

11/3/2020



Polymorphism

* Polymorphism means many forms,
* In java, the ability of a method to behave differently when different objects are acting upon it is called as Polymorphism
* In other words, a method exhibiting different forms when different objects are acting upon it is called Polymorphism
* polymorphism is categorized into 2 types
  1) Compile Time polymorphism
  2) Run Time polymorphism

Compile Time Polymorphism

* Compile Time polymorphism is achieved with the help of method overloading
* Compile Time polymorphism is also referred as static Binding / Early Binding.
* The decision of which method has to be executed is decided by compiler during compile Time.
* In so many overloaded method, the decision of which method implementation has to be executed is taken by the compiler during compile time
* In simple words, method binding happens during compile time by the compiler

Demo.java

Ex)

```
Public class Demo{
{
  void display (int a)
  {
    S.O.pln (a);
  }

  void display (String a, int b)
  {
    S.O.pln (a+ " "+b);
  }
```

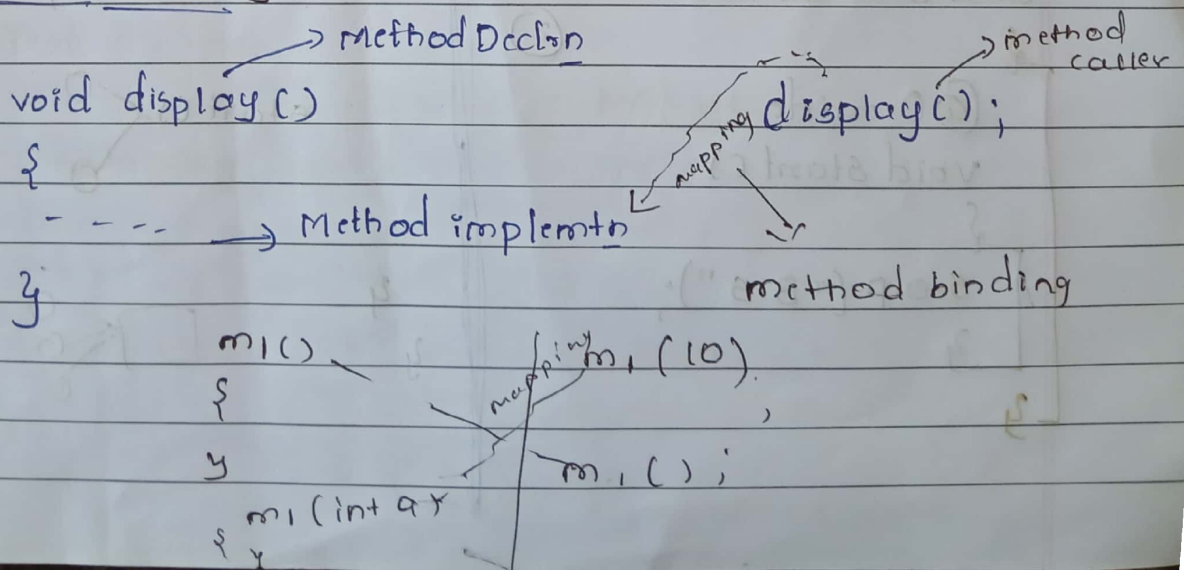
```
void display(double a)
 {
    s.o.pln(a);
 }

void display(int a, String b)
 {
   s.o.pln(a+" "+b);
 }

psvm(String[] args)
{
  Demo d = new Demo();
   d.display(23.45);
   d.display(100);
   d.display(10, "hai");
   d.display("bye", 1234);
   }
}
```

output

```
23.45
100
10 hai
bye 1234
```

Method Binding

Method Binding is a process of associating or mapping method callen to its method implementation is called as method Binding



JVM at runtime
Compiler at Compilation

Run Time Polymorphism

Run Time polymorphism can be achieved with the help of
1) IS-A-Relationship
2) Method Overriding
3) Upcasting

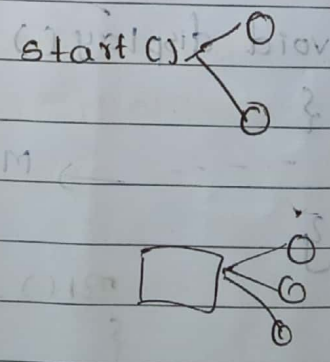
* Runtime polymorphism is also referred as late Binding or dynamic binding.
* During Runtime polymorphism, method binding always happens at runtime done by JVM
* When you call an overridden method on the superclass reference, the method implementation which gets executed depended on the subclass acting upon it
* The decision of which method has to get executed is taken by JVM at runtime.

Ex>

(1)
```
class Vehicle
{
  void start()
  {
    S.O.P("VS");
  }
}
```

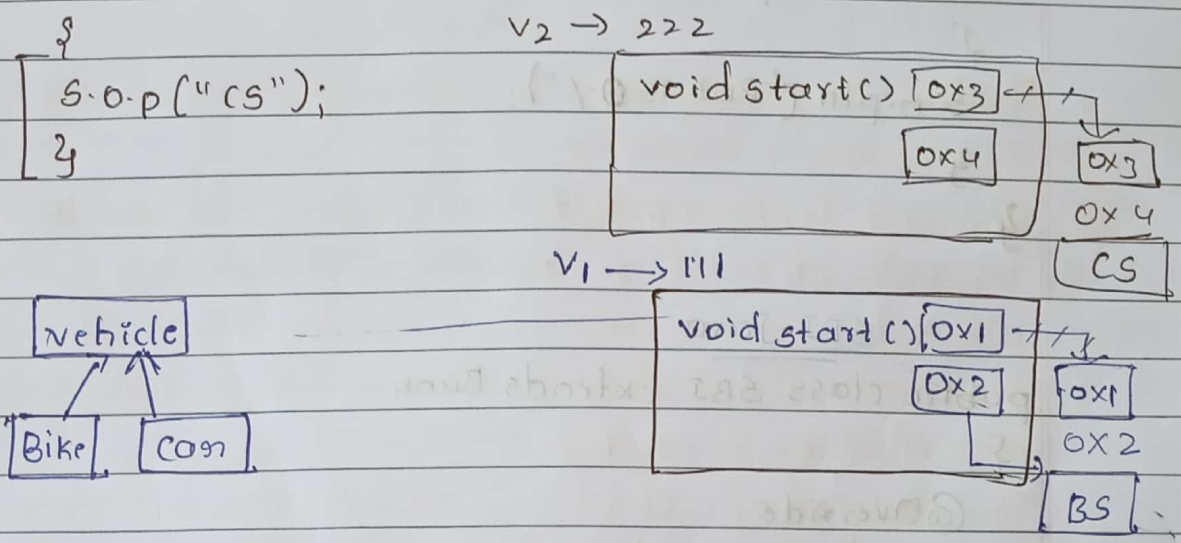
(2)
```
class Bike extends Vehicle
{
  @Override
  void start()
  {
    S.O.P("B.S");
  }
}
```

(4)
```
class Test
{
PSVM(---)
{
  Vehicle v1 = new Bike();
  v1.start();

  Vehicle v2 = new Car();
  v2.start();
}
}
```



③ class car extends Vehicle

```
{
   @Override
    void start()
    {
    S.O.p("CS");
    }
}
```



Ex> Another way of doing

```
class Test
{
 void display (vehicle v)
  {
    S.O.pl v.start();
  }

pvsM (_____)
{
 Test t = new Test();
  t.display (new Bike());  // BS
  t.display (new Car());   // CS
   Car c = new Car();
     or
  t.display (c);
}
}
```

Father s

Bank.java

```
Public class Bank
 {
 void rateOfInterest()
 {
  S.O.pln ("ROI is 0%");
 }
}
```

SBI.java

```
public class SBI extends Bank
 {
   @Override
    void rateOfInterest()
    {
      S.O.pln ("ROI is 6%");
    }
 }
```

ICICI.java

```
public class ICICI extends Bank
  {
    @Override
     void rateOfInterest()
     {
       S.O.pln ("ROI is 8%");
     }
}
```

customer.java

```
public class customer
{
  psvm (String [] args)
   {
    Bank b1 = new SBI();
    b1.rateOfInterest();

   Bank b2= new ICICI();
      b2.rateOfInterest();
   }
}
```

O/P

ROI IN SBI is 6%
ROI IN ICICI is 8%

or

```
public class AccountHolder
{
  void checkROI (Bank b)
   {
    b.rateOfInterest();
   }

  psvm (String [] args)
  {
   AccountHolder a = new AccountHolder();

    a.checkROI (new ICICI());
    a.checkROI (new SBI());

    S.O.pln(" ==--= ");

     SBI s = new SBI();
    a.checkROI(s);
```

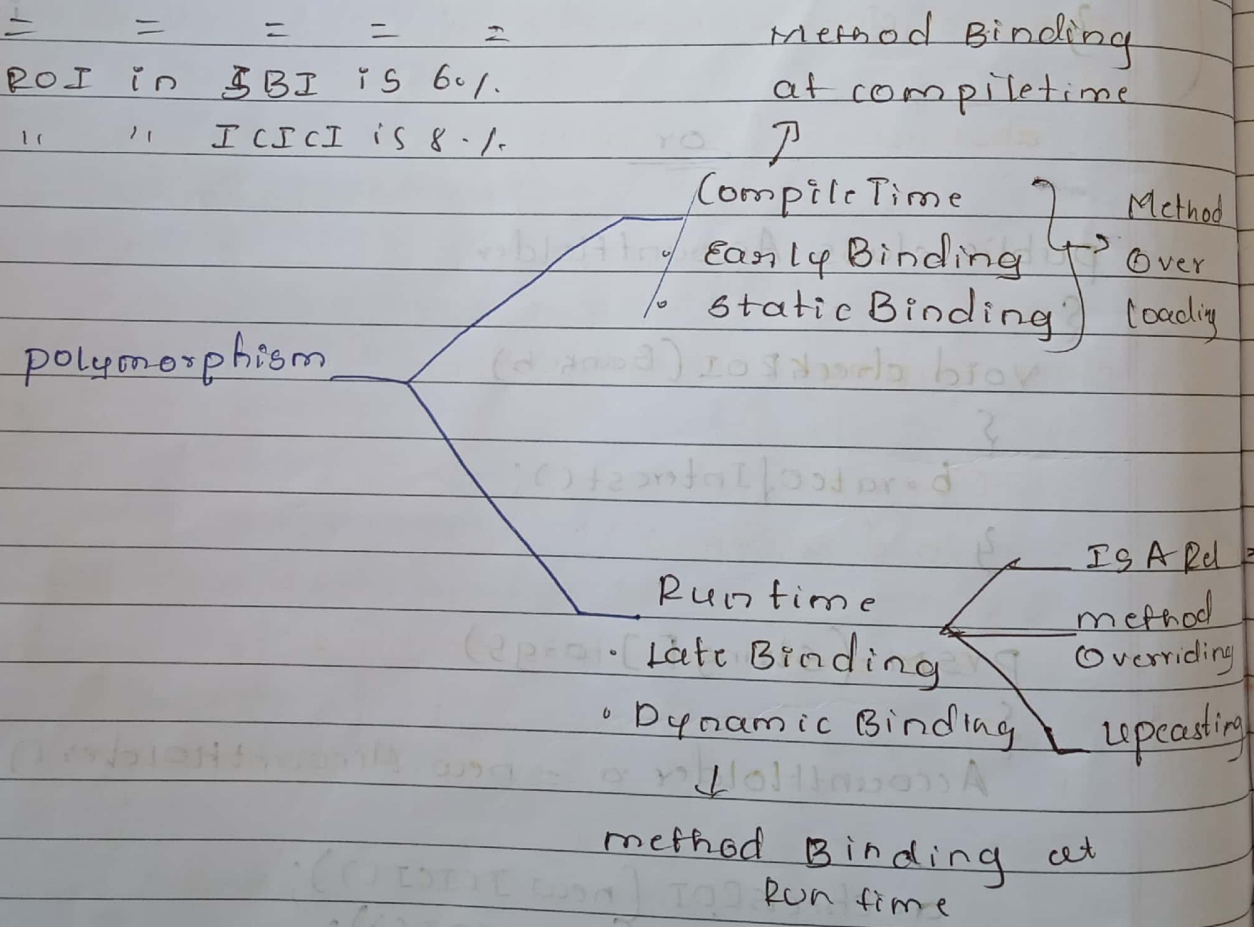
```
ICICI i = new ICICI();
  a.checkROI(i);
 }
}
```

O/P

ROI in ICICI is 8%.
" " SBI " 6%.

= = = = =

ROI in SBI is 6%.
" " ICICI is 8%.



12/3/2020

Ex)

12/3/2020

: Abstraction :

* Abstraction is a process of hiding the implementation details & showing only the Behaviour or the functionalities using abstract class / Interface
* Abstraction can be achieved using following rules
1) Abstract class / Interface
2) Is-A Relationship (Inheritance)
3) Method Overriding
4) Upcasting

Ex) Person.java

```
abstract class person
{
   abstract void work();
}
```

Employee.java

```
class employee extends person
{
  @Override
     void work()
    {
      s.o.pln("person is working");
    }
}
```

Test.java

```
public class Test
{
 pvsm(string[] args)
   {
     Employee e=new Employee() // 6
        e.work();
```

```
        Person p = new Employee();   // p is working
        p.work();                    // p is working
    }
}
```

Ex) Bank.java

```
public class Bank
{
    void deposit(int amount);
    void withdraw(int amount);
    void checkBalance();
}
```

ATM.java

```
public class ATM implements Bank
{
    int balance = 5000;

    @Override
    public void deposit(int amount)
    {
        S.o.pln("Depositing" + amount);
        balance = balance + amount; // balance += amount
        S.o.pln(amount + "Deposited Successfully");
    }

    @Override
    public void Withdraw(int amount)
    {
        S.o.pln("Withdrawing" + amount);
        balance = balance - amount; // balance -= amount
        S.o.pln(amount + "Dep Withdrawn Successfully
}
```

```
@Override
    void checkBalance()
    {
     S.o.pln("Available Balance is" +balance);
    }

}
```

AccountHolder.java

```
public class AccountHolder
{
 pvsm(String[] args)
 {
    Bank a = new ATM();
      a. checkBalance();

    S.o.pln("= = = =");

    a.deposit(500);
    a.checkBalance();

    S.o.pln("= = = =");

    a. withdraw(500);
     a.checkBalance();
 }
}
```

output

```
Available Balance= 5000
= = = = --
Depositing 500
500 Deposited successfully
 Available Balance is 500,
= = = = --
Withdrawing 500
500 withdrawn
    successfully
Available Balance
    = 500
```

package banking;

Ex) import java.util.Scanner;

```
public class Demo
{
pvsm (String [] args)
{
 Scanner s = new Scanner(system.in);

 While (true)
 {
  S.o.p ln ("1. Hello \t 2. Bye \t 3: Exit");
  s.o.pln ("Enter choice");
  int choice = s.nextInt();

  Switch (choice)
   {
  Case 1 :
        s.o.pln ("Hello");
        break;

Case 2 : s.o.pln ("Bye");
        break;

case 3 : s.o.pln ("Thank you!!!");
        system.exit(0); // Terminates & stops execution

default : s.o.pln ("Invalid choice");
        }

    s.o.pln (" - - - - - ");
  }
}
}
```

Output

```
1: Hello  2: Bye  3: Exit
Enter choice
1
Hello
- - - - - - - -
1. Hello 2. Bye 3: Exit
Enter choice
2
Bye
- - - - - - - - -
1. Hello 2. Bye 3. Exit.
Enter choice
4
Invalid choice
- - - - - -
1: Hello 2: Bye 3: Exit
Enter choice
3
Thank you !!!
```

Abstraction using factory design pattern

Rules :

1) Generalize all the behaviours of implementation classes & store it into an interface.
2) Create an object of implementation classes & store its address into an interface reference variable.
3) Use the interface reference variable in order to access the methods of the implementation classes

Ex) Factory :- Taxi.java

```
package factorydesign;

public interface Taxi
 {
    void startTrip();
    void endTrip();
 }
```

ola.java

```
public class ola implements Taxi
  {
    @Override
public void startTrip()
       {
       s.o.pln("started a Trip using ola App");

       }

Public void endTrip()
          {
          s.o.pln("started a Trip / cancelled using ola App");
          }
  }
```

uber.java

```
public class uber implements Taxi
{
  @Override
   public void startTrip( )
   {
    S.o.pln("started a Trip using uber App");
   }

@Override
    public void endTrip()
   {
    S.o.pln("Canceled a Trip using uber App");
   }
}
```

person.java

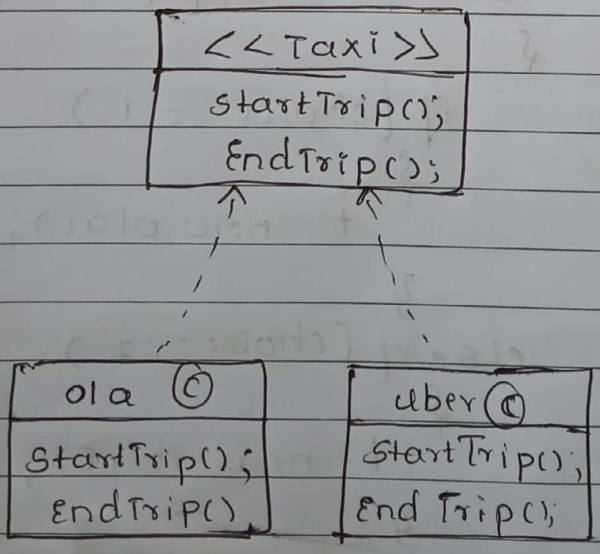
```
public class person
{
   Taxi t ;

  if (che    Taxi decideApp (int choice)
             {
                 if (choice == 1 )
                  {
                     t = new ola();
                  }
              else if (choice == 2 )
                  {
                     t= new uber();
                  }
                  return t ;
  }    }
```

Solution.java

```
public class solution
{
 pvsM (String[] args)
{
   person p = new person();

  Taxi t = p.decideApp();
      s.o.pln (t);
                 ola/uber address

     if (t! = null) } → to avoid nullpointer
     {                        exception
       t.startTrip()
       t.endTrip();

     }
}
```

output

```
factory design.ola@7852eaa2
Started a Trip using ola App
Cancelled a Trip using uber App
```



FoodDeliveryApp.java

```
public interface FoodDeliveryApp
{
    void orderFood();
    void cancelFood();
}
```

Swiggy.java

```
public class Swiggy implements FoodDeliveryApp
{
    @Override
    public void orderFood()
    {
        S.O.pln("ordering food using swiggy App")
    }

    @Override
    public void cancelFood()
    {
        S.O.pln("Canceled Food in swiggy APP");
    }
```

Zomato.java

```
public class Zomato implements FoodDeliveryApp
{
    @Override
    public void orderFood()
    {
        S.O.pln("ordering food using zomato Swiggy APP");
    }

    @Override
    public void cancelFood()
    { S.O.pln("canceling food using zomato APP"
```

foodpanda.java

```
public class foodpanda implements FoodDeliveryApp
{
 @Override
  public void orderFood()
   {

   }

 @Override
  public void cancelFood()
   {


   }
}
```

```
public class
 {
  FoodDelivery App F;

FoodDeliveryApp decideApp(int choice)
    {
      if (choice == 1)
      {
            F = new Swiggy();

      }

    else if (choice == 2)
      {
            F = new Zomato();
      }
```

mobile.java

```
public class mobile
{
  FoodDeliveryApp f;

FoodDeliveryApp decideApp (char choice)
    {
        if (choice == 's')
          {
            f= new swiggy();
          }

        elseif (choice == '2')
          {
            f= new zomato();
          }

        else if (choice == 'f')
          {
            f = new foodpanda();
}
```

```
else
{
S.o.pln("Invalid choice");
}

return f;
}
}
```

User.java

```
public class User
{
psvm(String[] args)
{
  Scanner s = new Scanner(System.in);
    Mobile m = new mobile();

S.o.pln("Enter choice:\n s:Swiggy\n z:Zomato\n
          f:foodpanda");

  Char choice = s.next().charAt(0);

FoodDeliveryApp f = m.decideApp(choice);

        if(f!=null)
        {
          f.orderFood();
          f.cancelFood();
        }
    }
}
```

Output

Enter your choice

1: Zomato

2: Swiggy

3: foodpanda

1

order a food by zomato

Cancel " " " "