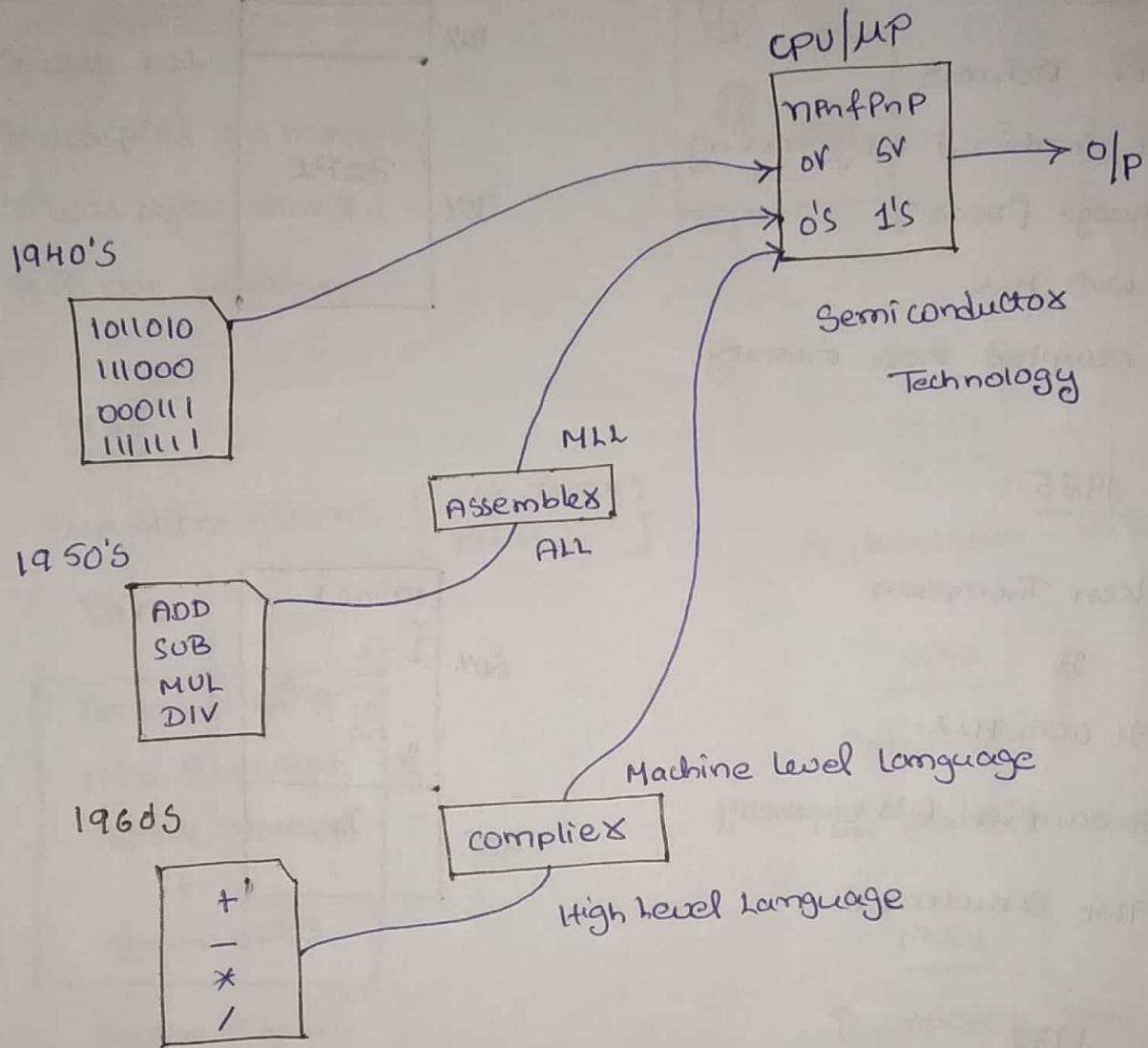


8/7/19

History of Programming Languages

→ 1945



→ Assembly is a software which converts Assembly level language into Machine level language.

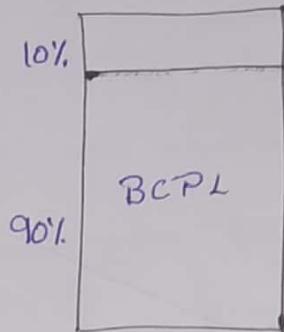
→ Compiler is a software which converts High level language into Machine level language.

9/14/2019

History of High Level Languages

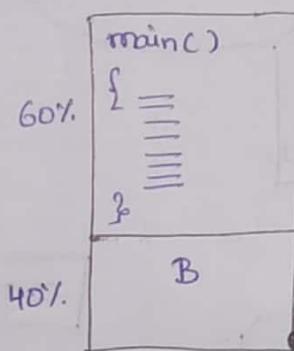
1962

- * Martin Richards
- * Basic combined programming language [BCPL]
- * It was HLL.
- * It occupied more memory.



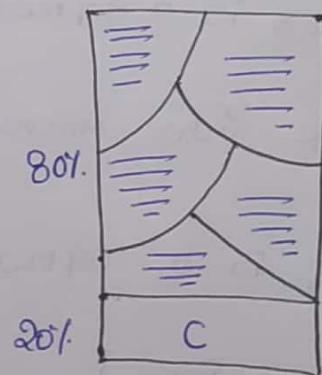
1965

- * Ken Thompson
- * B
- * It was HLL.
- * It occupied less memory.
- * not structured.



1972

- * Dennis Ritchie
- * C
- * It was HLL
- * It occupied less memory
- * It was Structured.
- * It was not object oriented.



1982

* Bjarne Stroustrup

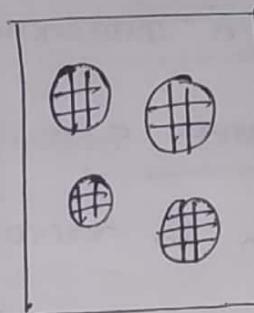
* C++

* It was HLL.

* It occupied less memory.

* It was object oriented

* It is not portable.

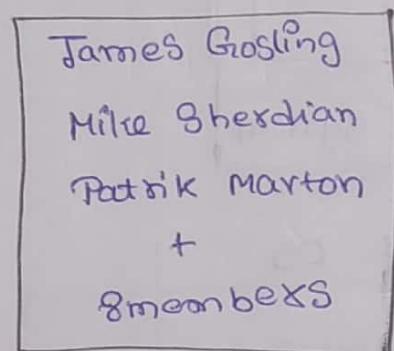


1985

* Sun Microsystems

{ oak, Green,
C+++- }

* Java



Green Team

1987

Afghanistan - Russia war

1990

Internet

Tim Berners Lee

1994

B-version Java

1996

Java 1.0

* It was HLL.

* It occupied less memory

* Object oriented

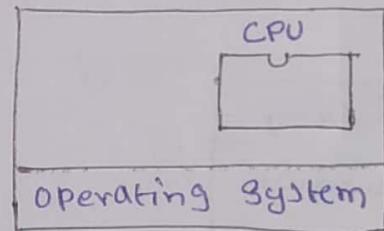
* Portable

Portability with respect to C++

Platform is a combination of most important hardware component and most important software component of a computer, which is microprocessor and operating system respectively. But, however the above given definition holds good in the case of a hardware engineer.

For software engineer platform means only and only operating system.

Portable
↔
platform independent



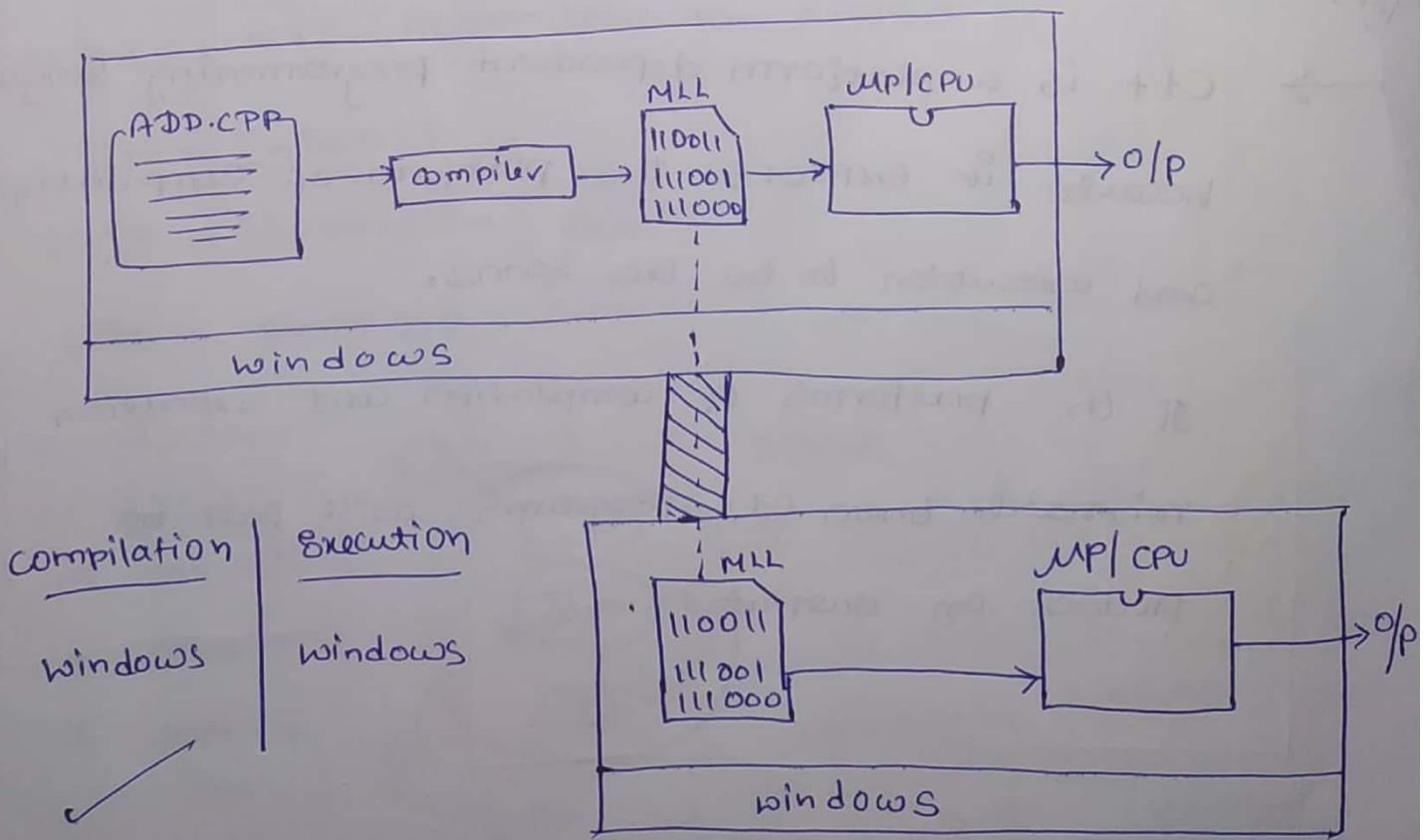
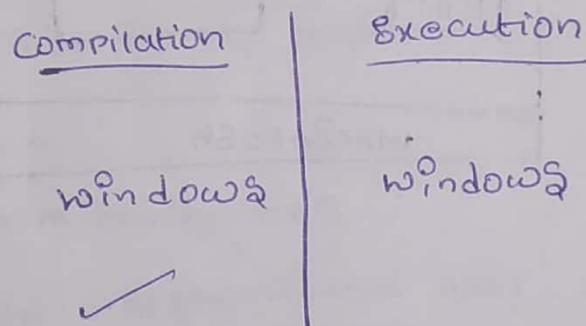
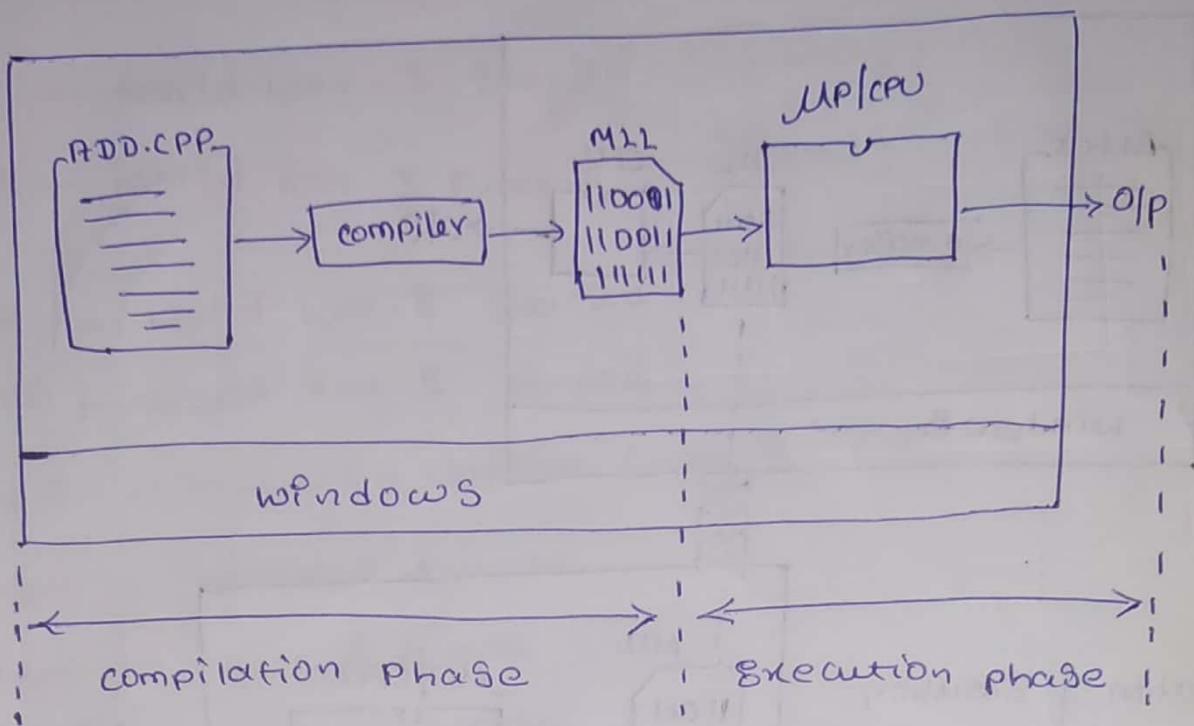
platform \Rightarrow CPU/MP + OS
[Hardware engineer]
i7 + windows
AMD + Linux
i7 + Macintosh

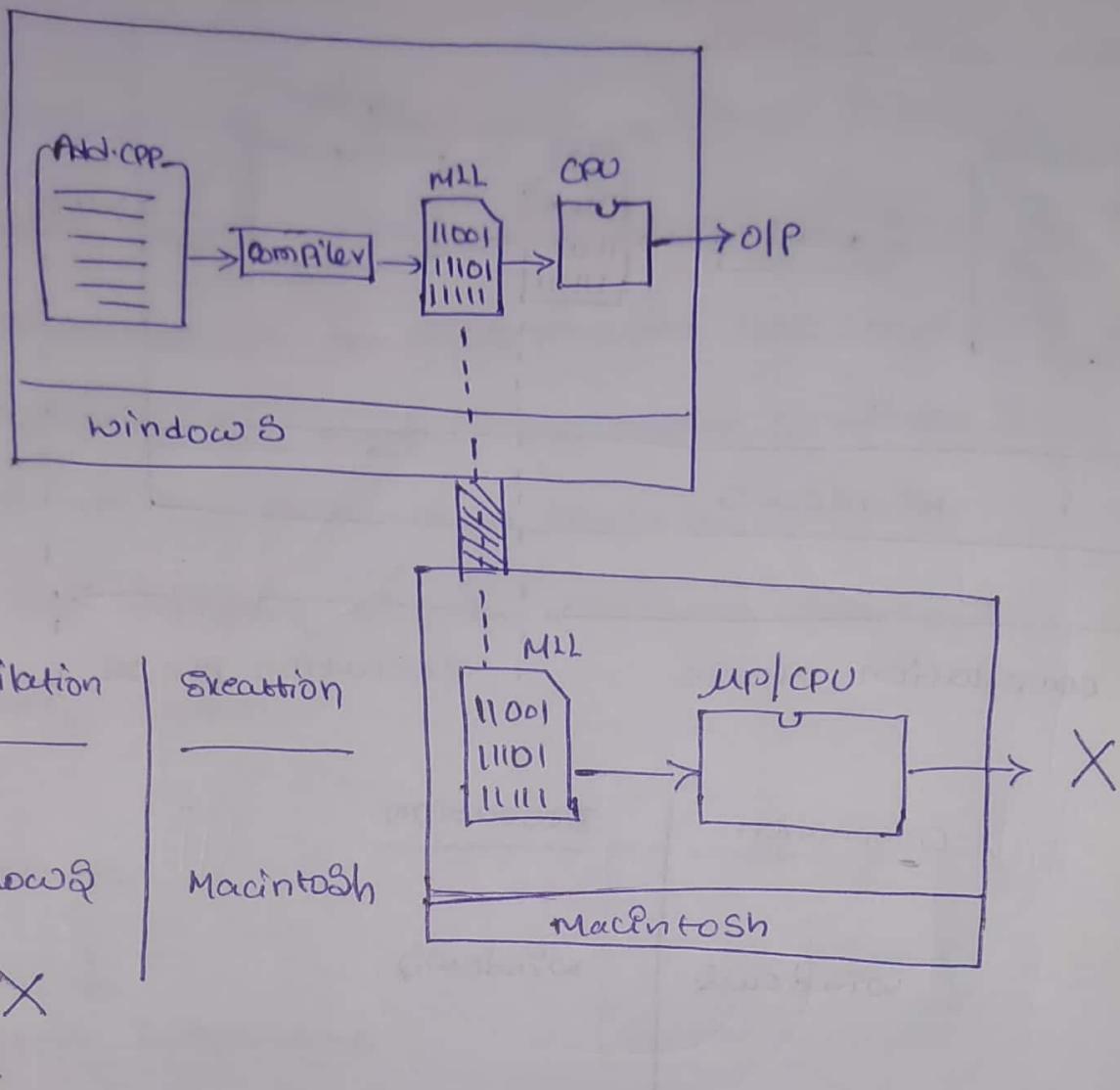
(Software Engineer) platform \Rightarrow OS

Windows

Linux

Macintosh





10/07/2019

→ C++ is a platform dependent programming language because it expects the platform of compilation and execution to be the same.

If the platforms of compilation and execution mismatch then C++ programs will fail to produce an output.

Historical Reasons Behind Java Success

1914 — World War-I Started [Germany]

X

1918 — World War-I Ended [Poland]

1939 — World War-II Started

1945 — World War-II Ended.

1945 — UNO - United Nations Organisation

1950's — Peaceful Decade

1960's — Peaceful Decade

1970's — Peaceful Decade

1980's — Two Super Powers [USA & USSR] (Cold War)

1982 — C++

1985 — Sun Microsystems

1989 — USSR — Afghanistan War Started

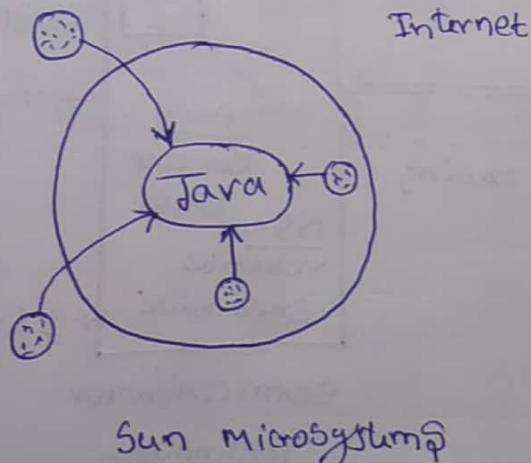
1989 — USSR — Afghanistan War Ended

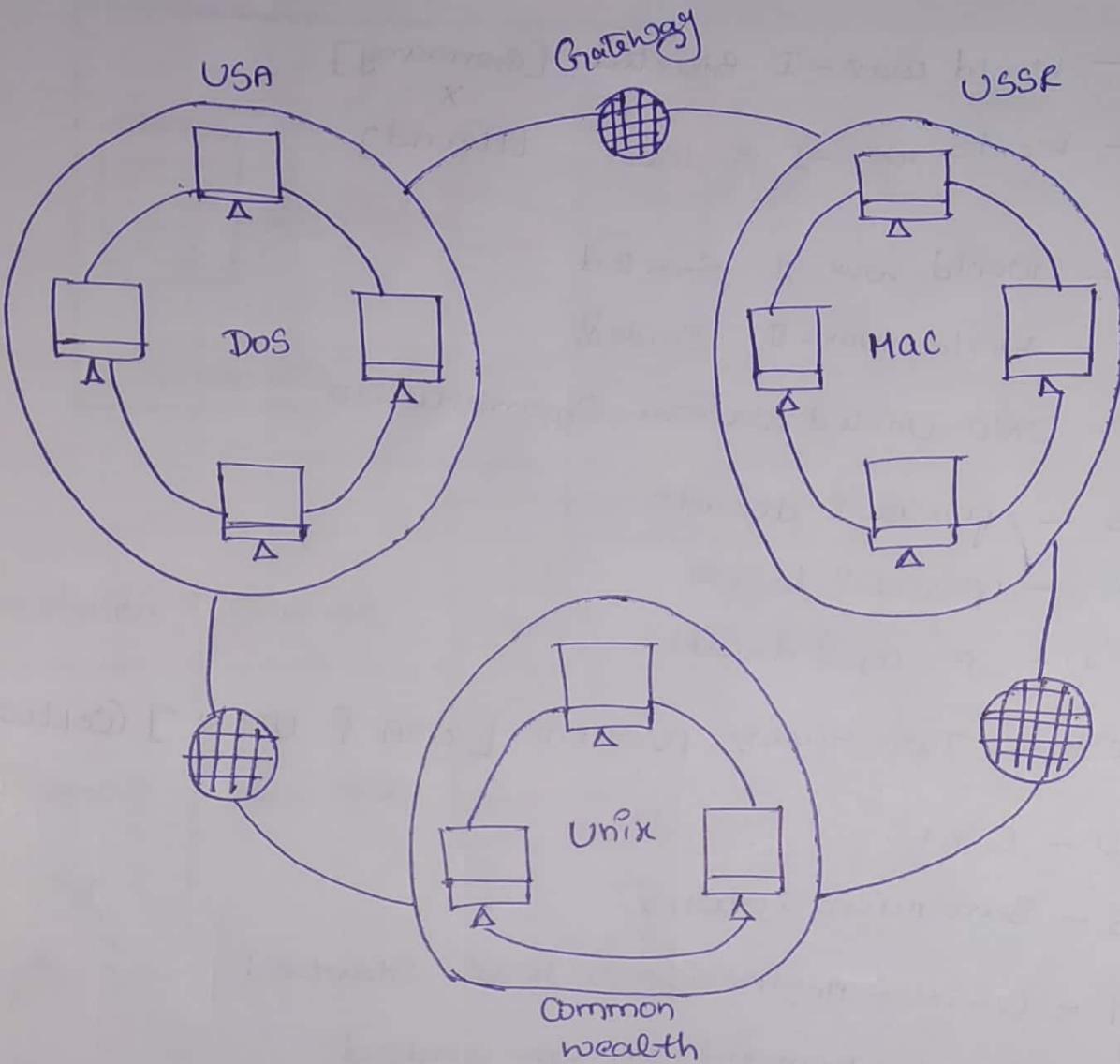
1990 — Internet

1994 — Beta Version Java

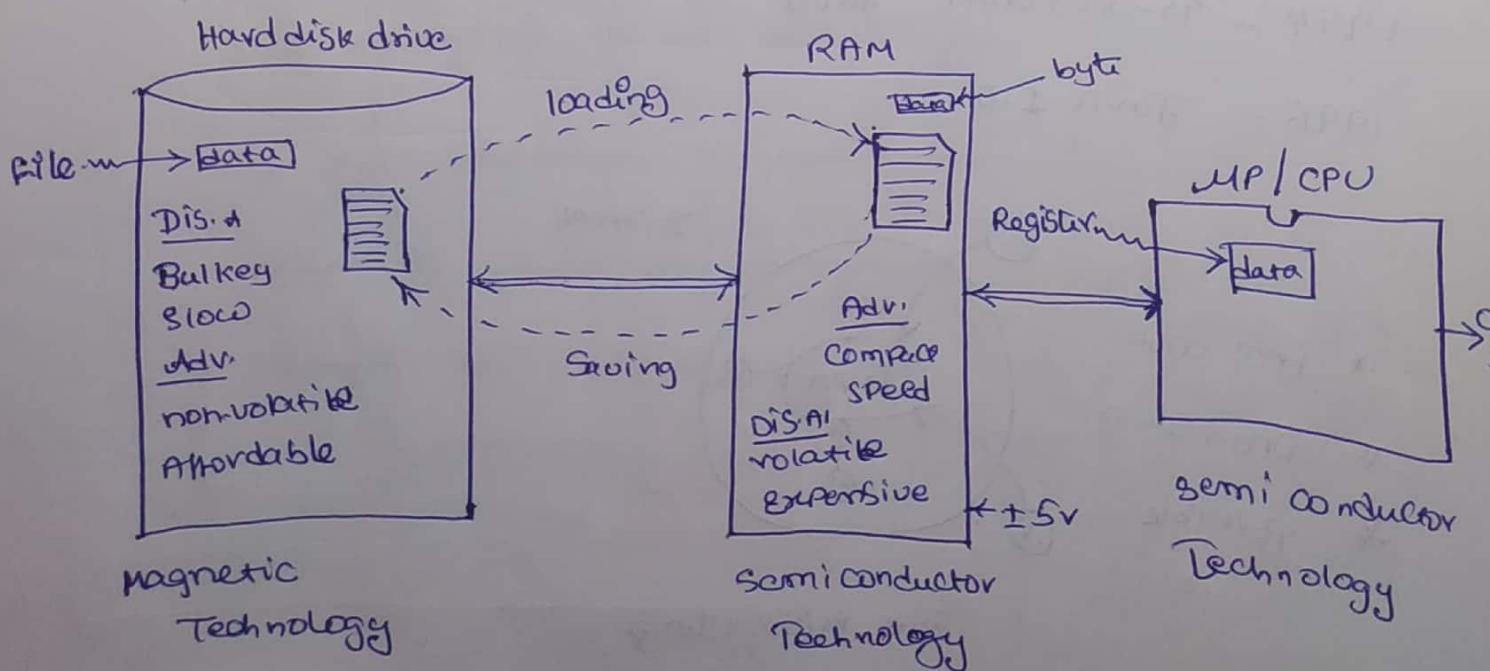
1996 — Java 1.0

- * Free Cost
- * Open Source
- * Portable





Organisation of Computer :-



loading :- It is a process of taking a copy of data present inside the hard disk and storing it inside the RAM.

Saving :- It is a process of taking a copy of data present in the RAM and storing it back into the Hard disk.

file :- It is a memory location available to the hard disk for the user to store data.

Byte :- It is a memory location available in the RAM for the user to store data.

Registers :- It is a memory location available to the micro processor / CPU for the user to store data.

→ We have four expectations from a memory device they are 1) Cheap / Affordable
2) Non-volatile
3) Fast
4) Compact

There is no one single memory device even today that can satisfy all the four expectations. Hence even today we used two memory devices hard disk and RAM.

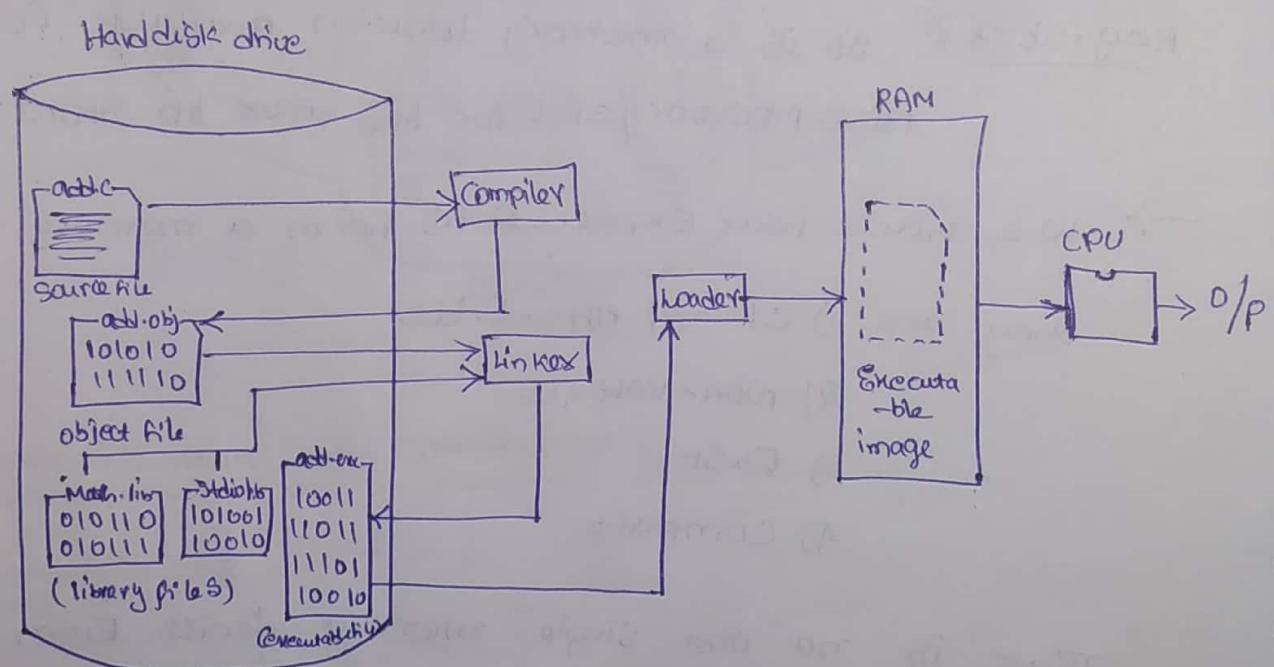
Object file Vs Executable file

Object file

Executable file

- | | |
|--|--|
| <ul style="list-style-type: none"> * It contains MLL code. * It is incomplete. * It cannot be executed. * It is small in size. | <ul style="list-style-type: none"> * It contains MLL code * It is complete. * It can be executed. * It is large in size. |
|--|--|

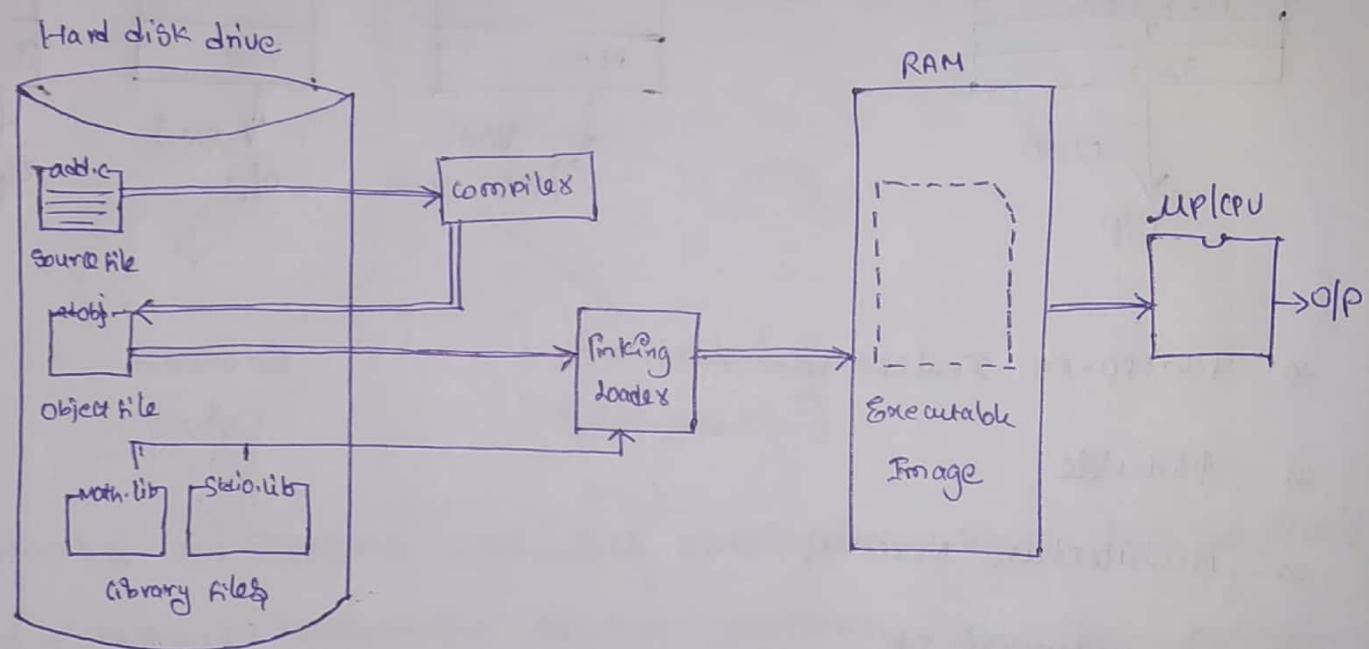
→ Source file is a file which contains HLL code.



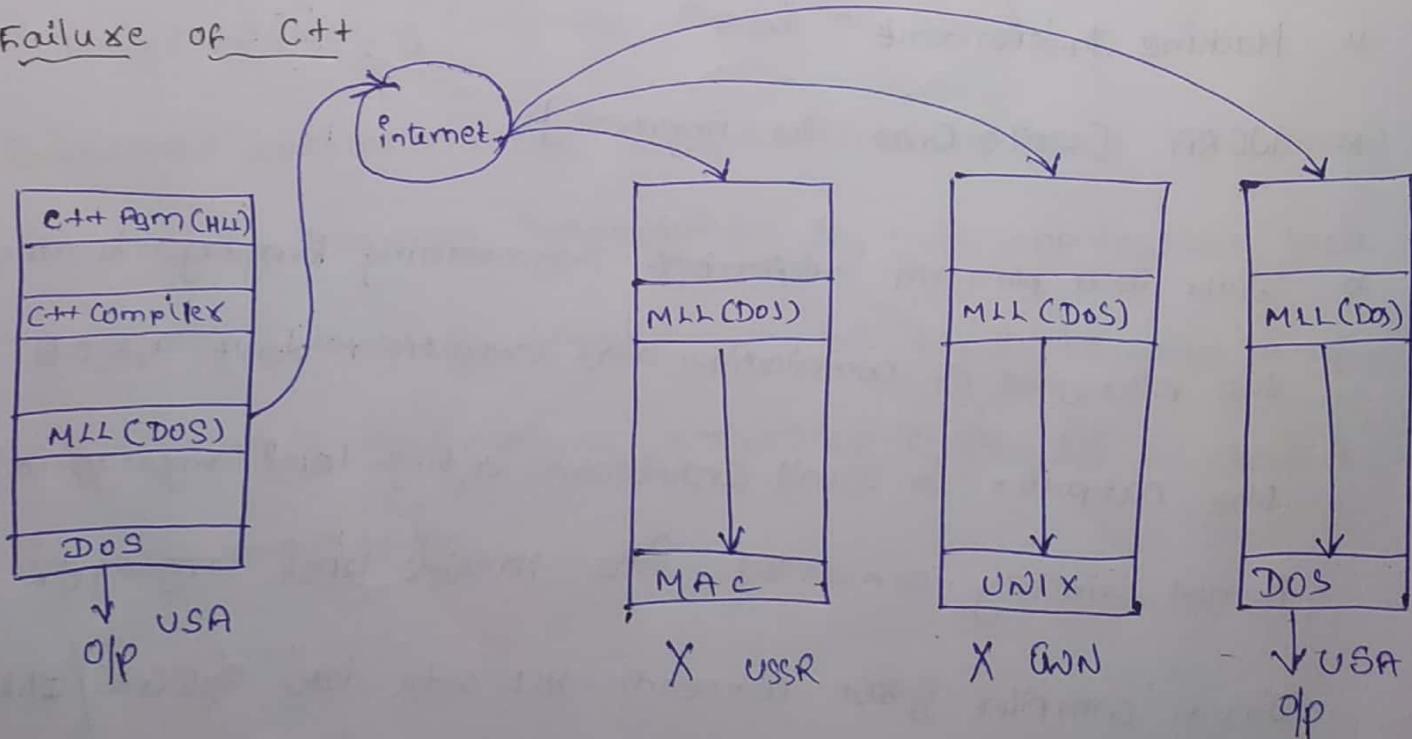
→ Linker is a software which takes object file and library files as an input and combines both to produce executable file.

→ Loader is a software which takes executable file as an input and loads it on to the RAM.

→ Linking Loader is a software which takes object file and library files as an input it produces an executable file but, however the executable file is not stored inside the hard disk rather it is directly loaded on to the RAM.

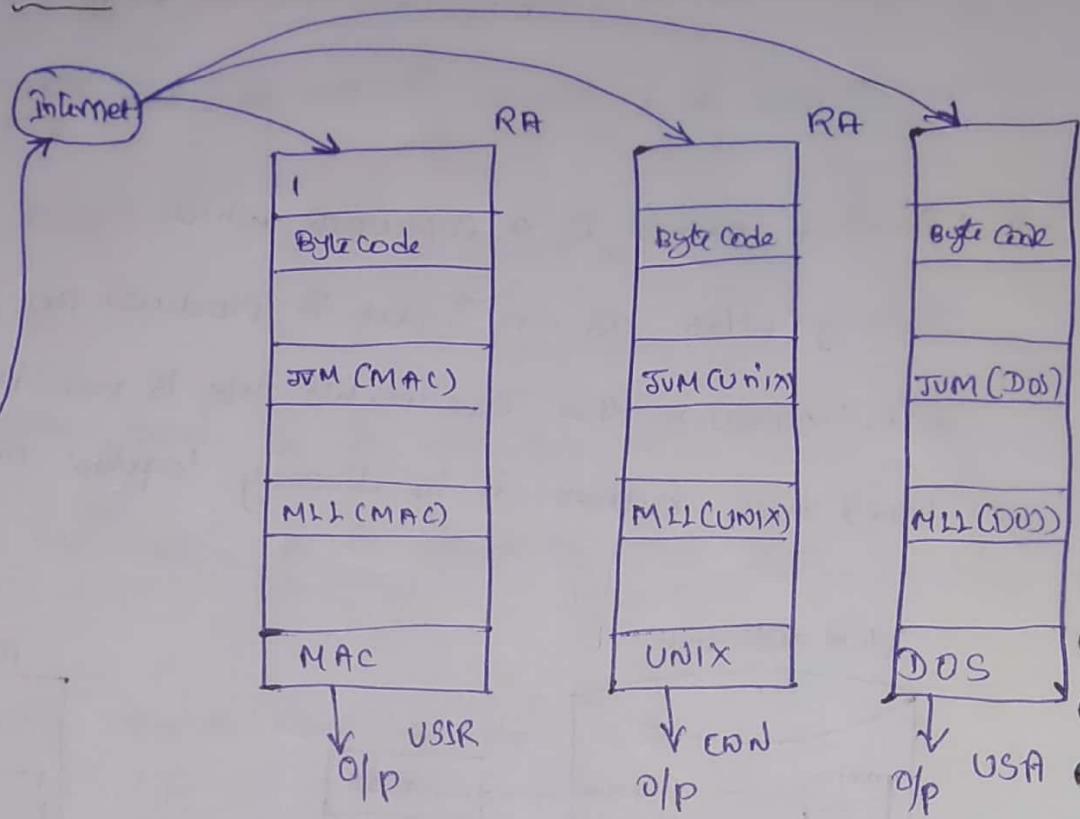
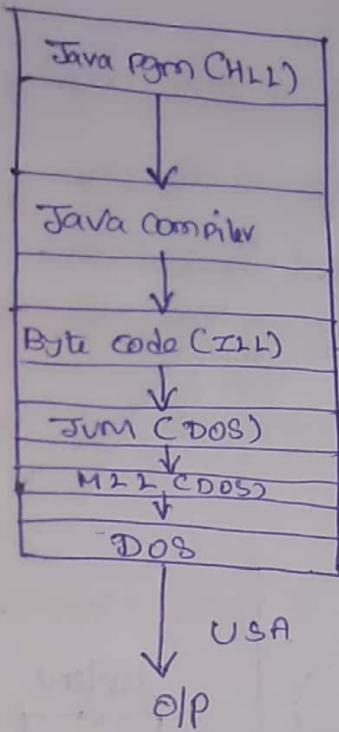


Failures of C++



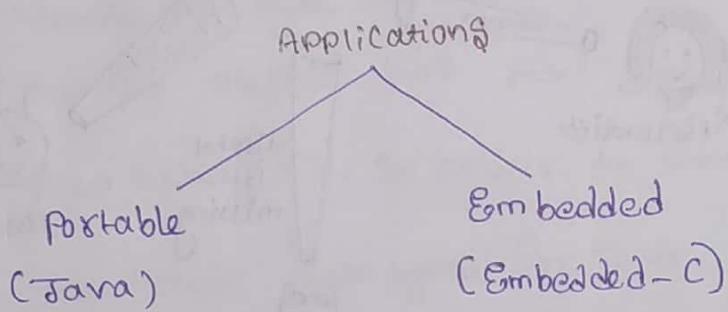
Success of Java :-

WORA



- * Platform Independent
- * Portable
- * Architecture Neutral
- * Os Independent
- * Machine Independent
- * WORA [Write Once Run Anywhere]
- * Java is a platform independent programming language because the platforms of compilation and execution don't affect the output. In Java's architecture a high level language code is not directly converted into machine level language.
- Java compiler first converts HLL code into Byte Code / JILL.

- * Later byte code is converted into MLL code using a Software called Java Virtual Machine (or) JVM.
- * JVM is a platform dependent Software because it is coded using 'c' programming language.
- * There are two major disadvantages of Java.
 - 1) Compare to 'c' programming language Java is slow in execution.
 - 2) Java is not a purely a object oriented programming language.



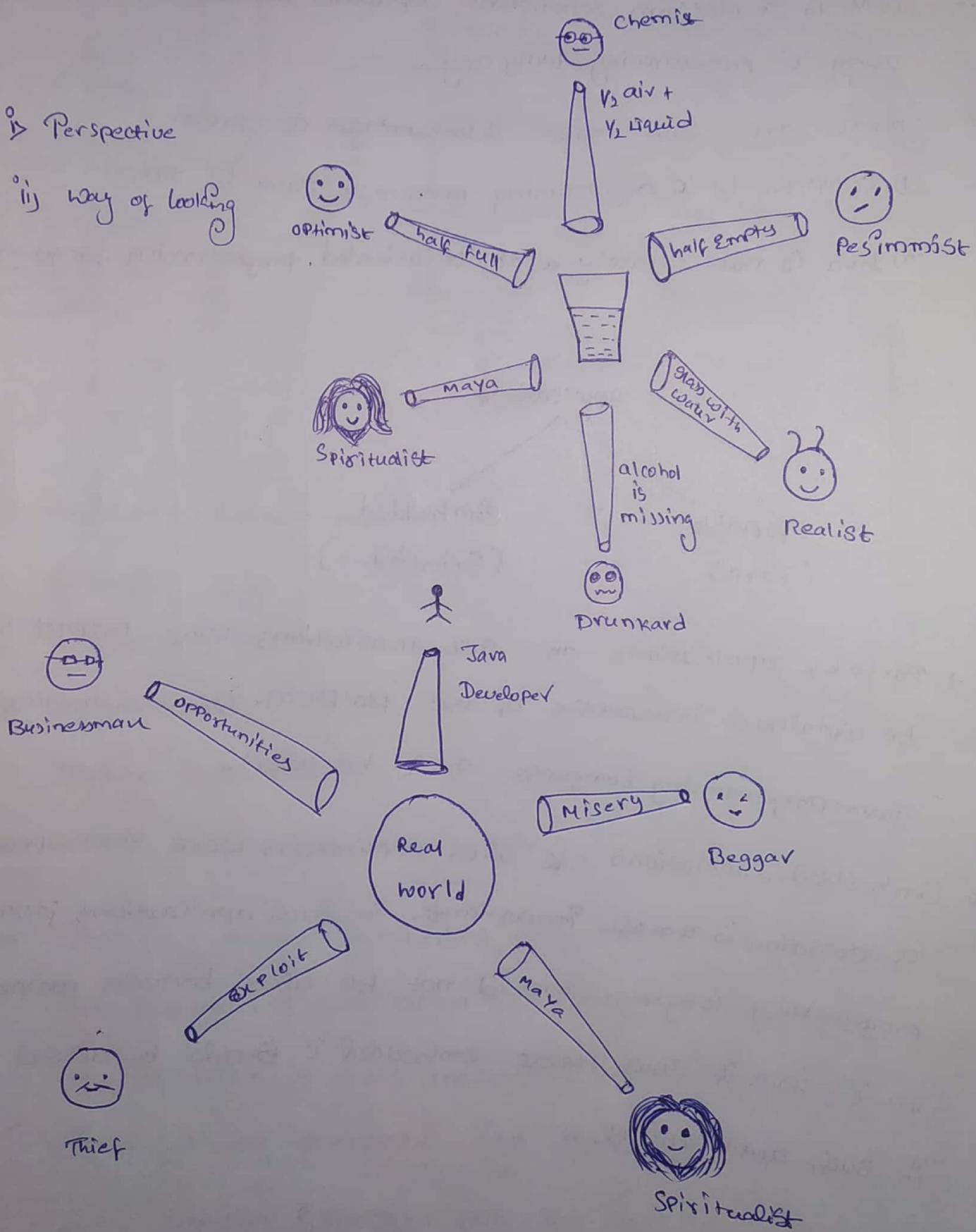
- * Portable applications are such applications where output should be obtained irrespective of the platform. For such applications Java programming Language should be used.
- * Embedded applications are such applications where the speed of execution is most important. In such applications java programming language should not be used because compare to 'c', Java is slow. Hence embedded 'c' should be used in such applications.

Object Orientation

Object Orientation is a perspective in which we look at the real world as collection of objects.

i) Perspective

ii) Way of looking



Rules of Object Orientation

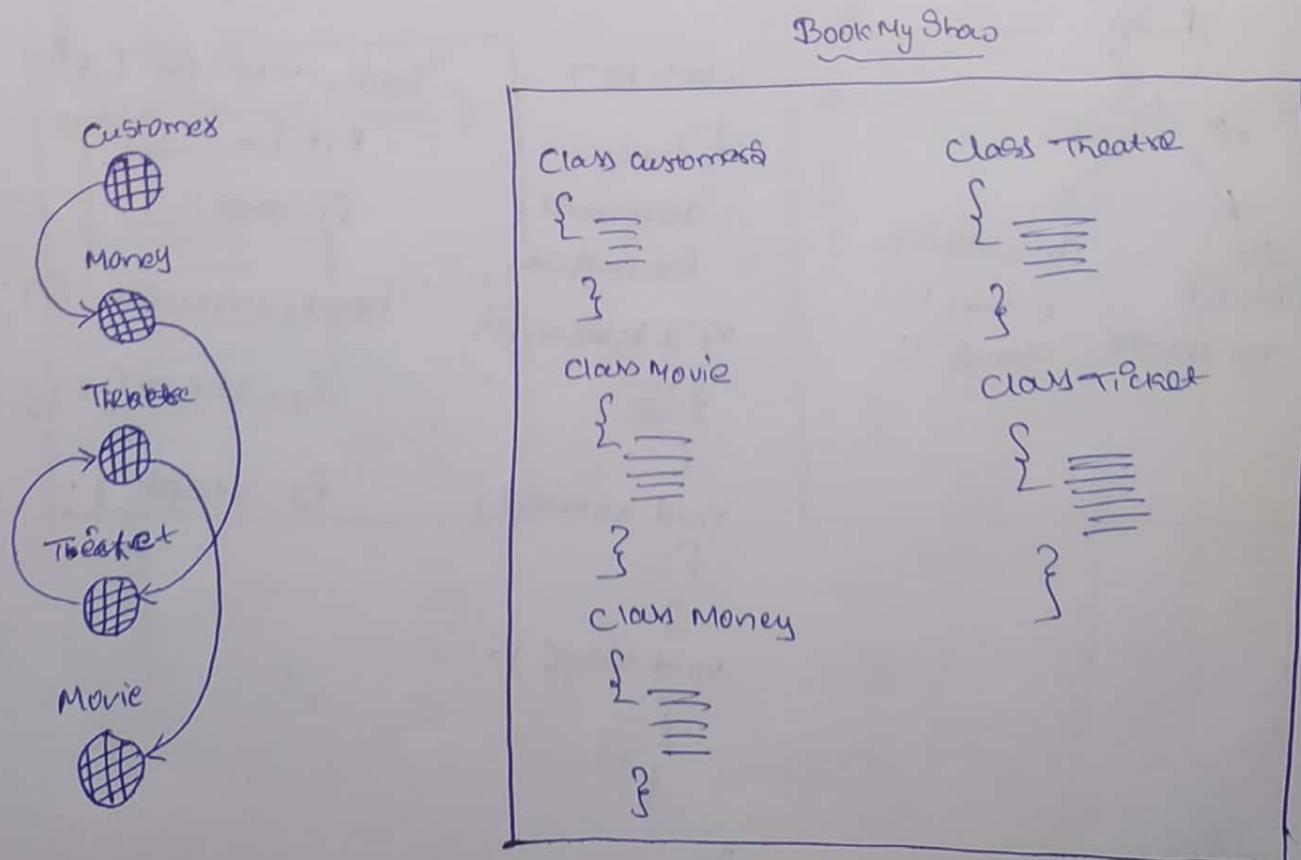
Rule 1 :- World is a collection of objects.

Rule-2: Every object is useful, no object is useless.

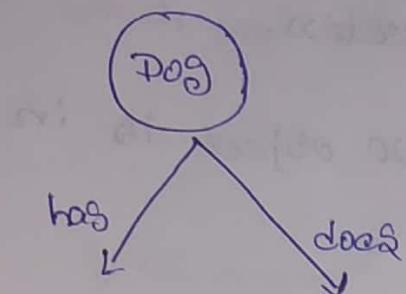
Rule-3:- Every object is in constant interaction, no object is in complete isolation.

Rule 4: Every object belongs to a type. Type doesn't exist in reality, what exists in reality is object. In order to handle the type we have 'class' in Java.

Rule 5:- Every object has few properties and does few behaviours. In order to handle the has part we have 'datatypes' in Java. In order to handle the does part we have 'methods' in Java.



Object Creation :



String name;
String breed;
float cost;

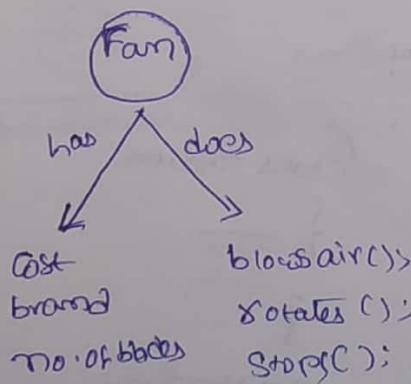
```

class Dog
{
    String name;
    String breed;
    String color;
    float cost;

    void eats();
    void barks();
    void sleeps();
}
  
```

Dog d = new Dog();
d.eats(); d
d.sleeps();

Dog d2 = new Dog();
d2.barks(); d2
d2.sleeps();



cost
brand
no.of blades

```

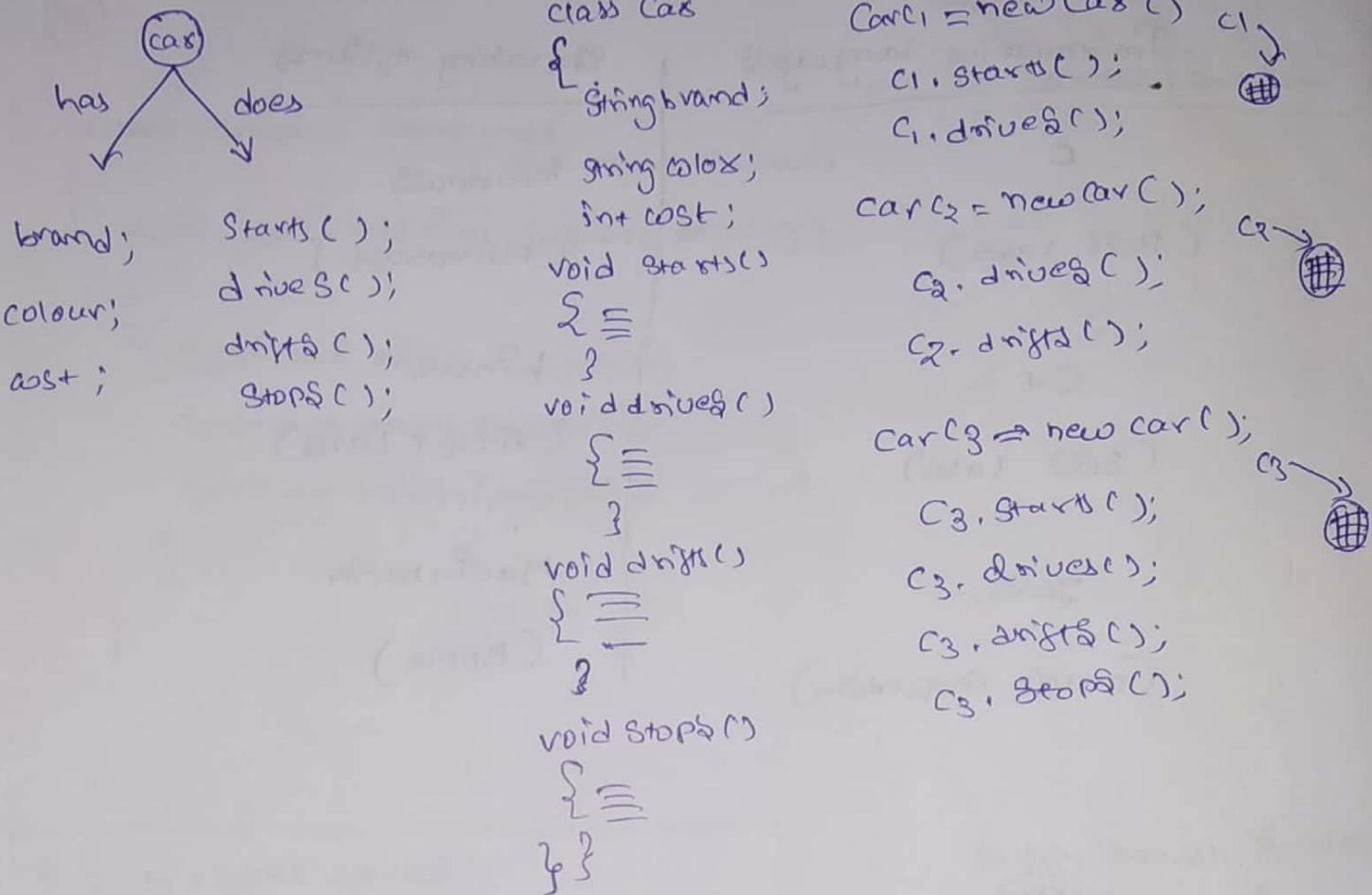
class Fan
{
    float cost;
    String brand;
    int no.of blades;

    void blows air();
    void rotates();
    void stops();
}
  
```

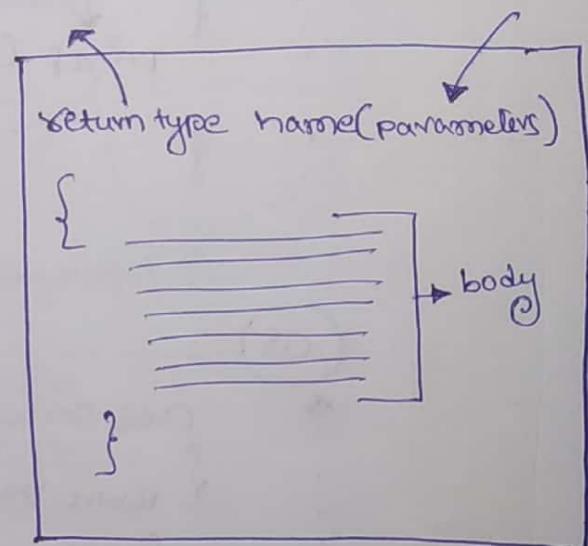
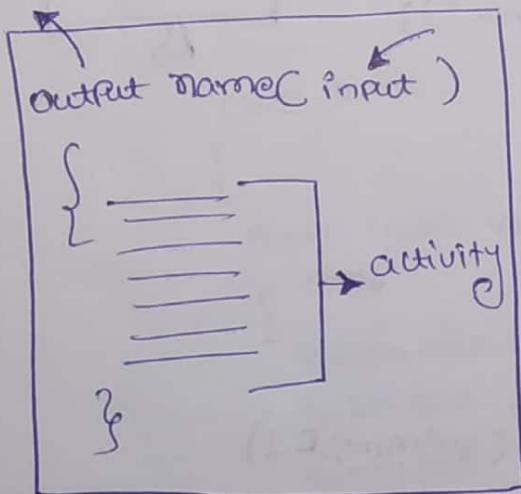
Fan f1 = new Fan();
f1.blows air();
f1.rotates();
f1.stops();

Fan f2 = new Fan();

f2.blows air();
f2.rotates();
f2.stops();



Main Method:



Programming Languages

C
(Bell Labs)

C++
(Bell Labs)

```

OS
+-----+
| void main ( ) |
| {           } |
|   printf ("Welcome to ABC"); |
| }           } |
+-----+
  
```

Operating Systems

Windows
(Microsoft)

Linux
(AT&T Labs)

Macintosh
(Apple)



```

OS
+-----+
| class Demo |
| {           } |
|   public static void main (String args [ ] ) |
|   {           } |
|     System.out.println ("Welcome to ABC"); |
|   }           } |
+-----+
  
```

Java



OS...

Class Demo

control + Data

[Sachin Ramesh Tendulkar]

Sachin Ramesh Tendulkar

args	0	1	2
	Sachin	Ramesh	Tendulkar

{ Public static void main (String args[])

{

System.out.println (args[0]);

System.out.println (args[1]);

System.out.println (args[2]);

}

}

E:\July>javac Demo.java

E:\July>java Demo Sachin Ramesh Tendulkar

Sachin Ramesh Tendulkar

args	0	1	2
	Sachin	Ramesh	Tendulkar

Command Line Arguments

Class Demo

{

Public static void main (String args[])

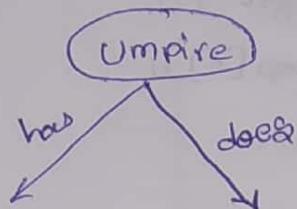
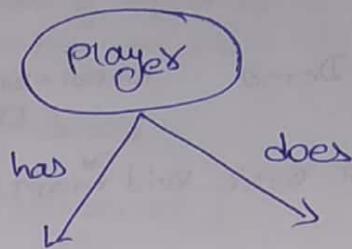
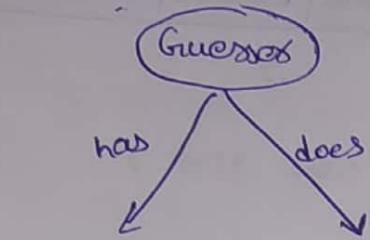
{

System.out.println (args[0]);

System.out.println (args[1]);

System.out.println (args[2]);

}



```

import java.util.Scanner;
class Guesses
{
    int gnum;

    int guessNum()
    {
        System.out.println ("Guesser please guess a number");
        Scanner scan = new Scanner (System.in);
        gnum = scan.nextInt();
        return gnum;
    }
}

```

A handwritten Java code snippet for a class named "Guesses". It contains a private integer variable "gnum" and a public method "guessNum()". The "guessNum()" method prints a message to the console, creates a new "Scanner" object from "System.in", reads an integer from the user using "nextInt()", and returns the value. The entire class definition is enclosed in curly braces at the bottom.

Class Player

```
{  
    int pnum;  
  
    int guessNum()  
    {  
        System.out.println("Player, please guess a number");  
        Scanner scan = new Scanner (System.In);  
        pnum = Scan.nextInt();  
  
        return pnum;  
    }  
}
```

Class Umpire

```
{  
    int numFromGuesser();  
    int numFromPlayer1;  
    int numFromPlayer2;  
    int numFromPlayer3;  
  
    void collectNumFromGuesser()  
    {  
        Guesser g = new Guesser();  
        numFromGuesser = g.guessNum();  
    }  
}
```

```
void collectNumFromPlayer()  
{
```

```
    Player p1 = new Player();  
    numFromPlayer1 = p1.guessNum();  
  
    Player p2 = new Player();  
    numFromPlayer2 = p2.guessNum();  
  
    Player p3 = new Player();
```

num from player 3 = P3.guessNum();

}

void compare()

{

if (num from guesser == num from player 1)

{

S.O.P ("Player 1 wins");

}

else if (num from guesser == num from player 2)

{

S.O.P ("Player 2 wins");

}

else if (num from guesser == num from player 3)

{

S.O.P ("Player 3 wins");

}

else

{

S.O.P ("Game Lost!! Try Again");

}

}

Class Launch

{

public static void main (String args[])

{

Umpire U = new Umpire();

U.collectNumFromGuesser();

U.collectNumFromPlayer();

U.compare();

}

}

Naming Convention in Java

Naming convention for variables :-

while declaring a variable all the letters should be in lower case.

Legal

```
int gnum;
int enum;
int x;
int y; ✓
```

illegal

```
int gnun;
int-Pnum;
int X; X
int Y;
```

Naming convention for Methods :- The name of the method should be a verb and during creation of the method the uppercase and lowercase combinations should be as shown below examples.

Legal

```
void compare()
int guessNum() ✓
void collectNumFromGuesser()
void collectNumFromPlayer()
```

illegal

```
void Compare()
int GuessNum()
void CollectNumFrom guaser()
void CollectNumFrom Player()
```

X

Naming convention for classes :- The name of the class should be a noun!

legal

```
Class Umpire ✓
class PassengerPlane
```

illegal

```
class Umpire X
class passengerPlane
```

Pattern Programming :-

```
int a=5;
    a [ 5 ]
++a;
S.O.P(a); // 6
```

```
int a=5;
    a [ $ 6 ]
++a;
S.O.P(a); // 6
```

```
int a=5;
int b=$
    a [ 5 6 ]
b=++a;
    b [ 6 ]
S.O.P(a); // 6
S.O.P(b); // 5
```

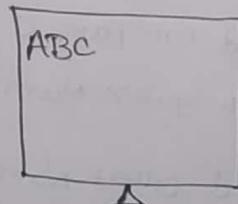
```
int a=5;
int b;
    a [ $ 6 ]
b=a++;
    b [ 5 ]
S.O.P(a); // 6
S.O.P(b); // 5
```

Difference b/w "print" and "println".

→ Class Launch

{

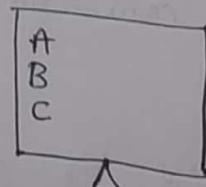
```
public static void main (String args[])
{
    System.out.print("A");
    System.out.print("B");
    System.out.print("C");
}
```



→ Class Launch

{ public static void main (String args[])

```
{
    System.out.println("A");
    System.out.println("B");
    System.out.println("C");
}
```



Loops :-

- i) Initialization
- ii) Condition Check
- iii) Incrementation/ Decrementation

* For Loop

* While Loop

* Do-While loop

for loop

```
for (initial, condi, inc/dec)
{
}
```

while loop

```
initialization
while(condition)
{
    inc/dec
}
```

do-while loop

```
initialization
do
{
    incr/decr
} while(condition);
```

→ Class Launch

```
{
    public static void main (String args[])
    {
        System.out.print ("*");
    }
}
```

O/P - *

For loop:

Class Launch

```
{
    public static void main (String args[])
    {
        for (int i=1; i<=5; i++)
        {
            System.out.print ("*");
        }
    }
}
```

O/P - * * * * *

While loop

Class Launch

```
{
    public static void main (String args[])
    {
        int i=1;
        while(i<=5)
        {
            System.out.print ("*");
            i++;
        }
    }
}
```

O/P - * * * *

do-while loop

Class Launch

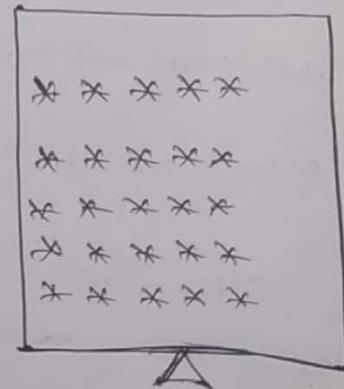
```
{
    public static void main (String args[])
    {
        int i=1;
        do
        {
            System.out.print ("*");
            i++;
        } while(i<=5);
    }
}
```

O/P - * * * *

Pattern - 1

Class Launch

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



Pattern - 2

class Launch

{

```
public static void main(String args[])
```

{

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

{

```
for (int k = 1; k <= 5; k++)
```

```
{ System.out.print("* "); }
```

{

```
for (int j = 1; j <= 5; j++)
```

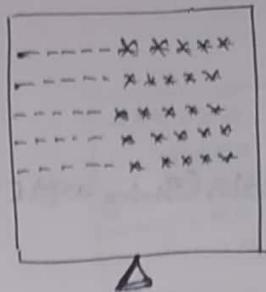
```
{ System.out.print("*"); }
```

{

```
System.out.println();
```

}

}



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

Pattern - 3

class Launch

```
public static void main(String args[])
```

{

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

{

```
for (int j = 1, j <= i; j++)
```

{

```
System.out.print("*");
```

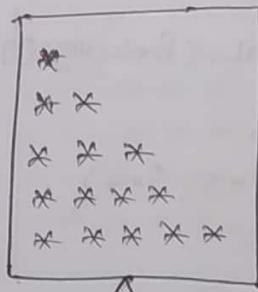
{

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

{

}

}



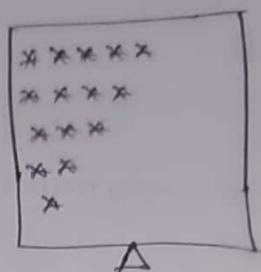
i	j
1	1
2	2
3	3
4	4
5	5

Pattern - 4

Class Launch

```

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



i	j	(5-i+1)
1	5	(5-1+1)
2	4	(5-2+1)
3	3	(5-3+1)
4	2	(5-4+1)
5	1	(5-5+1)

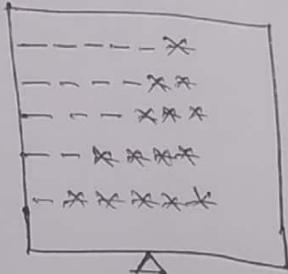
i k j
1 5 1
2 4 2
3 3 3
4 2 4
5 1 5

Pattern - 5

Class Launch

```

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

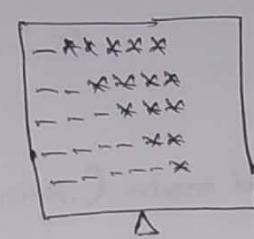


i k j
1 5 1
2 4 2
3 3 3
4 2 4
5 1 5

Pattern - 6

Class Launch

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

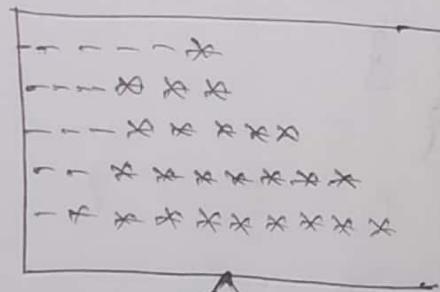


i	j	k
1	1	5
2	2	4
3	3	3
4	4	2
5	5	1

Pattern - 7

Class Launch

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



i	j	k
1	1	1
2	3	2 * i - 1
3	5	3
4	7	4
5	9	5

$$\underline{[5-i+1]}$$

Pattern - 8

Class Kisan

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

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

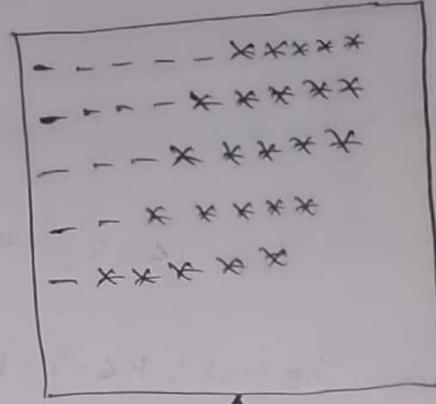
```
{ for (int j=1; j<=6-i; j++) }
```

```
{ System.out.print(" "); }
```

```
for (int k=1; k<=5; k++)
```

```
{ System.out.print("*"); }
```

```
System.out.println();
```

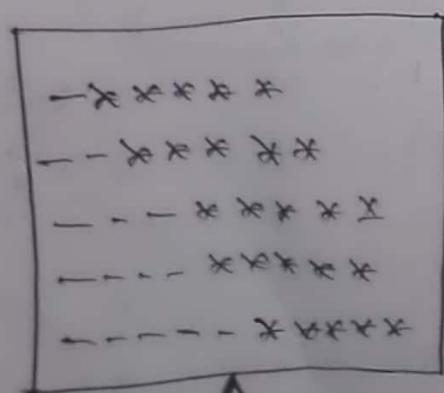


i j k

1	5	5
2	4	5
3	3	5
4	2	5
5	1	5

$5-i+1$

Pattern - 9

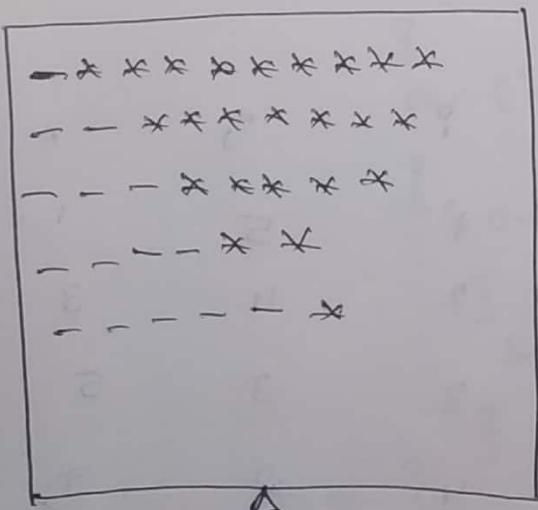


1	1	5
2	2	5
3	3	5
4	4	5
5	5	5

Class Haxi

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

Pattern - 10



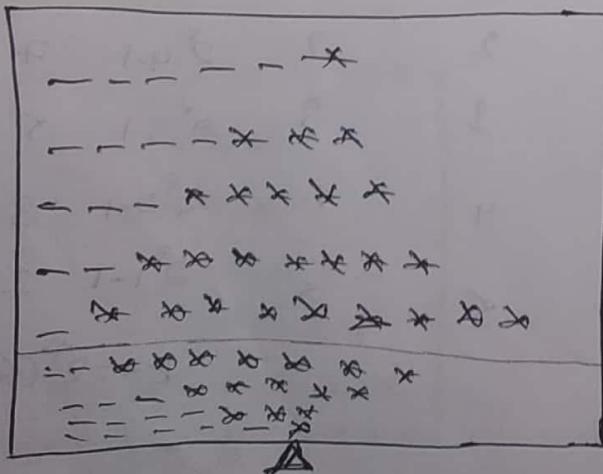
i	j	k	
1	1	$2^5 - 1$	9
2	2	$2^4 - 1$	7
3	3	$2^3 - 1$	5
4	4	$2^2 - 1$	3
5	5	$2^1 - 1$	1

$k \leq 2^{(G-i)} - 1$

Class Hanoi

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

Pattern - 11



i	j	k
1	5	1
2	4	3
3	3	5
4	2	7
5	1	9

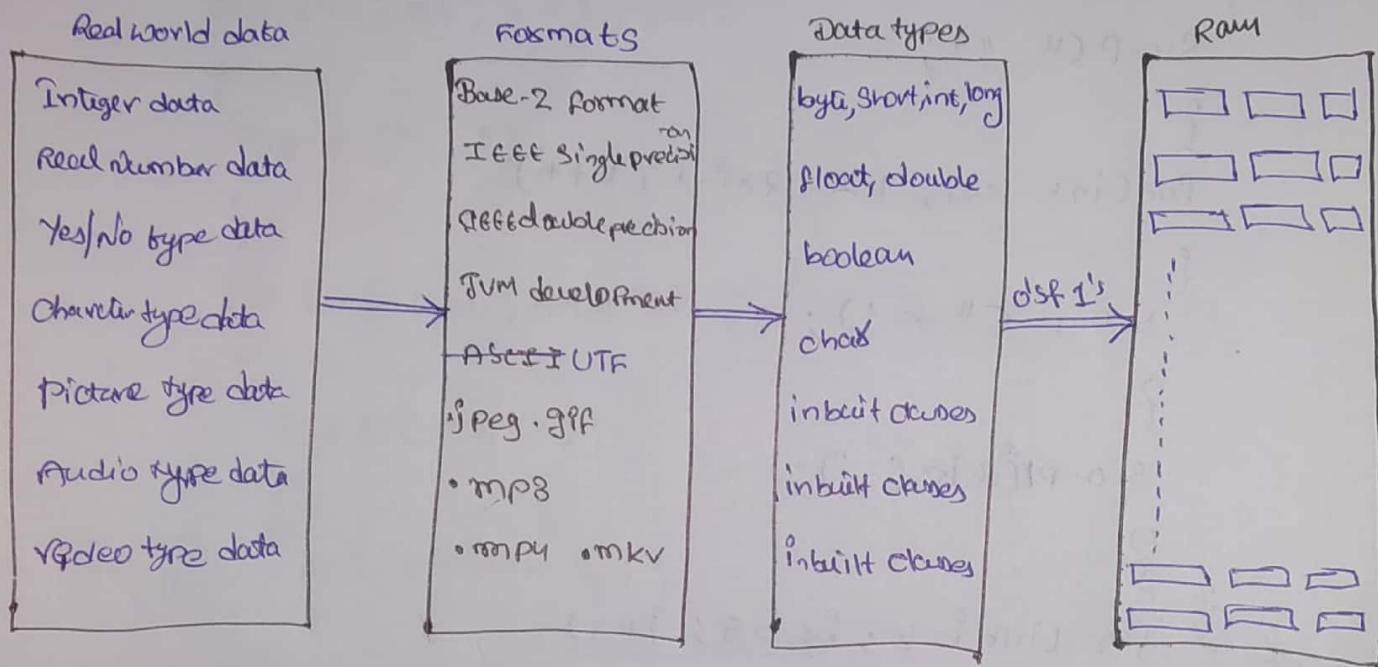
$$j = 6-9 \quad k = 2-i-1$$

Class Hanoi

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

Data types

Data types are the facilities provided by programming language in order to convert the real world data into computer understandable data as 0's and 1's.



2 GB

2×10^9 bytes

$2 \times 2 \times 10^9$ bits

32×10^9 transistors

32,000,000,000 transistors.

Semi conductor
Technology

Integer datatype :-

byte

short

int

long

byte a;



1 byte

base@-2 formula

$$-2^{n-1}$$

to

$$+2^{n-1} \text{ (no. of bits = } n)$$

$$-2^{8-1}$$

to

$$+2^{8-1}$$

$$-2^7$$

to

$$+2^7$$

$$-128$$

to

$$+127$$

Eg:

Age

Room temperature.

Short a;



~~short~~ 2-bytes

$$-2^{16-1}$$

to

$$+2^{16-1}$$

$$-2^{15}$$

to

$$+2^{15}$$

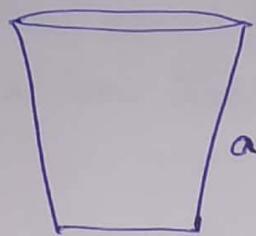
$$-32,768 \text{ to } +32,767$$

Eg:

Average salary of fresher

Population of a locality,

int a;



Eg.:

populations of city C

→ Distance b/w two countries.

4 bytes

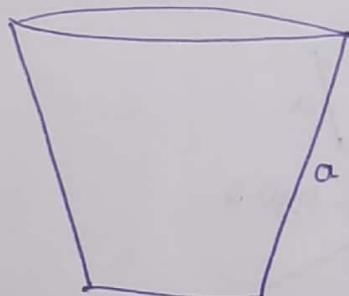
-2^{n-1} to $+2^{n-1}$

-2^{32-1} to $+2^{32-1}$

-2^{33} to $+2^{33}$

-2147483648 to +2147483647

long a;



Eg.:

Account number

→ distance b/w two planets.

-2^{64-1} to $+2^{64-1}$

-2^{63} to $+2^{63}-1$

-92233720368547758082 to 92233720368547758071

* $\log_{10} a = 45;$

$$\begin{array}{r} 2 \overline{|} 45 \\ 2 \overline{|} 22 - 1 \\ 2 \overline{|} 11 - 0 \\ 2 \overline{|} 5 - 1 \\ 2 \overline{|} 2 - 1 \\ 1 - 0 \end{array}$$

a	0	1	2	3	4	5	6	7
	0	0	1	0	1	1	0	1

↑
MSB
[Most significant bit]
LSB
[Least significant bit]

MSB decides the
Sign of integers

0 → +ve

1 → -ve

* byte $a = -45;$

$$\begin{array}{r} 2 \overline{|} 45 \\ 2 \overline{|} 22 - 1 \\ 2 \overline{|} 11 - 0 \\ 2 \overline{|} 5 - 1 \\ 2 \overline{|} 2 - 1 \\ 1 - 0 \end{array}$$

a	0	1	2	3	4	5	6	7
	1	1	0	1	0	0	1	1

↑
MSB

00101101

11010010 1's compliment

+1 2's compliment

11010011

Prefixes that can be attached to integers Literal

`int a=45;`

`S.O.P(a);`

`O/p → 45`

`int a=045;`

`S.O.P(a);`

`O/p → 37
(Octal)`

`int a= 0x45;`

`S.O.P(a);`

`O/p → 69
(Hexa decimal)`

`int a= 0b101101;`

`S.O.P(a);`

`O/p → 45`

`(binary)`

Q-7

0	000	0 4 5	0 x 4 5
1	001	/ \	/ \
2	010	1 0 0	0100
3	011	1 0 1	+ 2^6 + 2^2 + 2^0
4	100	2^5 2^2 2^0	
5	101		
6	110		
7	111	<u>37</u>	<u>69</u>

Class Demo

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

```
{ int a=099; // 10011001
```

```
System.out.println (a);
```

```
}
```

Note :-

- * There are four different data types present in java in order to convert integer type data into 0's and 1's. They are i) Byte, ii) Short, iii) Int and iv) Long.

We have '4' different data types because in the real world integer data exists in different magnitudes. Hence in order to efficiently utilise the memory we have four data types.

- * If '0' is applied as a prefix to Integer literal the integer literal will be stored in octal format.

- * if '0x' is applied as a prefix to integer literal then the integer literal will be stored in Hexadecimal format.
- * if '0b' is attached as a prefix to integer literal then the integer literal is stored in the form of binary.

Real numbers Data types :

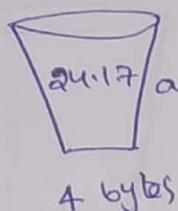
i) float

float a = 24.17f;

ii) double

1	8	23
0	10000011	100000101.....

sign Exponent Mantissa
 $(8 \times 2^8 - 127)$



$$\begin{array}{r} 2|24 \\ 2|12 \\ 2|6 \\ 2|3 \\ \cdot \end{array} \begin{array}{r} -8 \\ -0 \\ -0 \\ -1 \end{array}$$

$$\underline{11000} - 24$$

$$\begin{array}{r} 0.17 \times 2 = 0.34 \\ 0.34 \times 2 = 0.68 \\ 0.68 \times 2 = 1.36 \\ 1.36 \times 2 = 0.72 \end{array} \begin{array}{r} 0 \\ 0 \\ 1 \\ 0 \end{array}$$

$$0.72 \times 2 = 1.44 \quad 1$$

$$0.44 \times 2 = 0.88 \quad 0$$

$$0.88 \times 2 = 1.76 \quad 1$$

$$0.76 \times 2 = 1.52 \quad 1$$

$$1.52 \times 2 = 1.04 \quad 1$$

$$1.04 \times 2 = 0.08 \quad 0$$

24.17
 \downarrow
 $11000 \cdot 0010101110$

1.10000010101110 (no. of shifts = 4)

1.10000010101110
 \downarrow
 Mantissa

$$\text{Exponent} = \text{no. of shifts} + 127$$

$$= 4 + 127$$

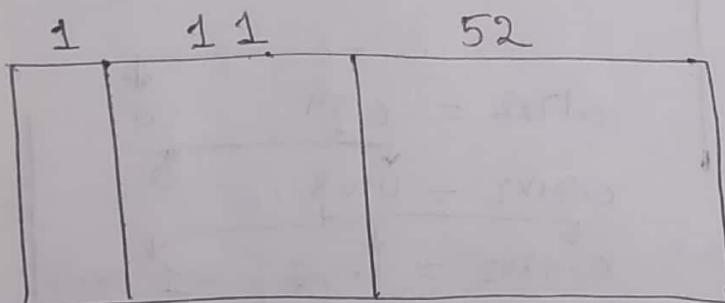
$$= 131$$

$$= \boxed{1000\ 0011}$$

↓
Exponent

$$\begin{array}{r}
 2 \overline{) 131} \\
 2 \overline{) 65} -1 \\
 2 \overline{) 32} -1 \\
 2 \overline{) 16} -0 \\
 2 \overline{) 8} -0 \\
 2 \overline{) 4} -0 \\
 2 \overline{) 2} -0 \\
 1 -0
 \end{array}$$

* double a;



Sign Exponent Mantissa

(excess-1023)

* There are two data types present in order to handle real number data they are 'float' and 'double'.

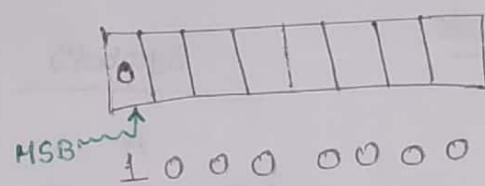
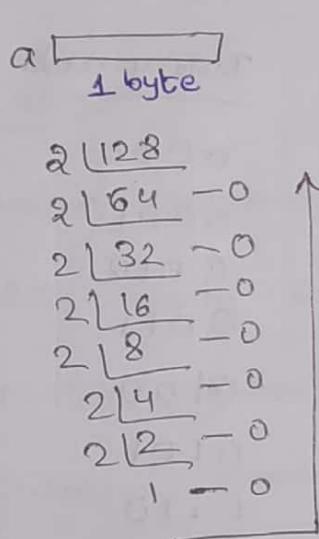
Based on the application one of them can be chosen. For high precision & high accuracy we should opt for double. If the application demands less precision

them we should choose float.

Note :-

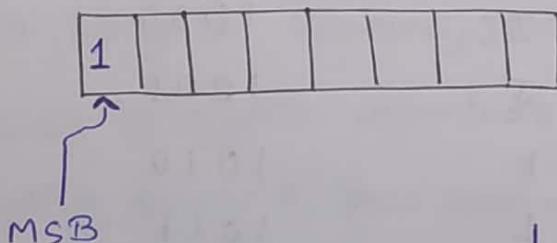
- * If a variable is declared as byte then only and only one byte of data is given for the program to store the data.
- * Hence the numbers ranging from '-128' to '+127' can be stored. But storing a value greater / Lesser than this range is not possible as shown below.

byte a = 128;



(Since it is a +ve number '0' is stored in MSB
hence the ^{no} memory is available for 8
more bits)

byte a = -128;



$$\begin{array}{r} 1000 \quad 0000 \\ \hline 0111 \quad 1111 \end{array}$$

i's compliment

$$\begin{array}{r} +1 \\ \hline 1000 \quad 0000 \end{array}$$

2's compliment

Character Data type :-

(8 4 2 1)
1 0 0 1, so

<u>Symbols</u>	<u>Binary code</u>
A	0
B	1

2 symbols $\rightarrow 2^1$
↓
no. of bits

<u>Symbols</u>	<u>Binary code</u>
A	0 0
B	0 1
C	1 0
D	1 1

4 symbols $\rightarrow 2^2$
↓
no. of bits

<u>Symbols</u>	<u>Binary code</u>
A	0 0 0
B	0 0 1
C	0 1 0
D	0 1 1
E	1 0 0
F	1 0 1
G	1 1 0
H	1 1 1

8 symbols $\rightarrow 2^3$
↓
no. of bits

<u>Symbols</u>	<u>Binary code</u>
A	0 0 0 0
B	0 0 0 1
C	0 0 1 0
D	0 0 1 1
E	0 1 0 0
F	0 1 0 1
G	0 1 1 0
H	0 1 1 1
I	1 0 0 0
J	1 0 0 1
K	1 0 1 0
L	1 0 1 1
M	1 1 0 0
N	1 1 0 1
O	1 1 1 0
P	1 1 1 1

16 symbols $\rightarrow 2^4$
↓
no. of bits

1st symbol----- 0 0 0 0 0 0 0 0

2nd symbol----- 0 0 0 0 0 0 0 1

A ----- 0 1 0 0 0 0 0 1

B ----- 0 1 0 0 0 0 1 0

Penultimate symbol --- 0 1 1 1 1 1 1 0

Ultimate symbol ----- 0 1 1 1 1 1 1 1

ASCII:-

- * American Standard Code for Information Interchange.
- * ASCII totally consists of 128 symbols and the binary code for 128 symbols is a '7' bit code. But, however the minimum memory given by the RAM is 1 byte / 8 bits. Hence, ASCII is forcefully stored as a 8 bit code even if it is a '7' bit code. However, Java doesn't follow ASCII because ASCII is English oriented + it has binary code for only English symbols, few special characters and numbers.
- * The binary code for the symbols which are largely used in other languages is not available. This is the disadvantage of ASCII.

1st symbol - - - - - 0000 0000 0000 0000
 end symbol - - - - - 0000 0000 0000 0001
 !
 !
 A - - - - - 0000 0000 0100 0001
 B - - - - - 0000 0000 0100 0010
 !
 !
 !
 !
 # - - - - - 0000 1001 0000 0101
 @ - - - - - 0000 1100 1000 1010
 !
 !
 !
 !
 65,535th symbol - - - - - 1111 1111 1111 1110
 65,536th symbol - - - - - 1111 1111 1111 1211

Universal Transfer Format → 16

UTF → 16

Unicode

65,536 → 2¹⁶
 no. of bits.

In Java in order to store a character 'char' datatype is used and internally 'char' datatype makes use of universal transfer format to convert the characters into 0's and 1's.

ASCII is not used, because in UTF we have 65,536 symbols that is, binary code is available for almost all largely spoken languages across the world. Since, 65,536 symbols are present 16 bits of memory is occupied by every character.

Note:

A real number literal in Java will automatically treated as a double value, so that data can be stored with high precision.

* double a = 38.17;

System.out.println(a);

O/p → 38.17

Hence, as shown above a real number literal can be directly stored inside double type variable without any error. But, if the real number literal is try to store inside a float type variable directly it results in an error. Because a float type variable is allocated just 4 bytes.

* float a = 38.17; //---- Error

S.o.p(a);

Hence, In order to resolve this error a suffix "f" should be attached to the real number literal (or) the real number literal should be type casted into float as shown below.

* float a = 38.17f;

float a = (float) 38.17;

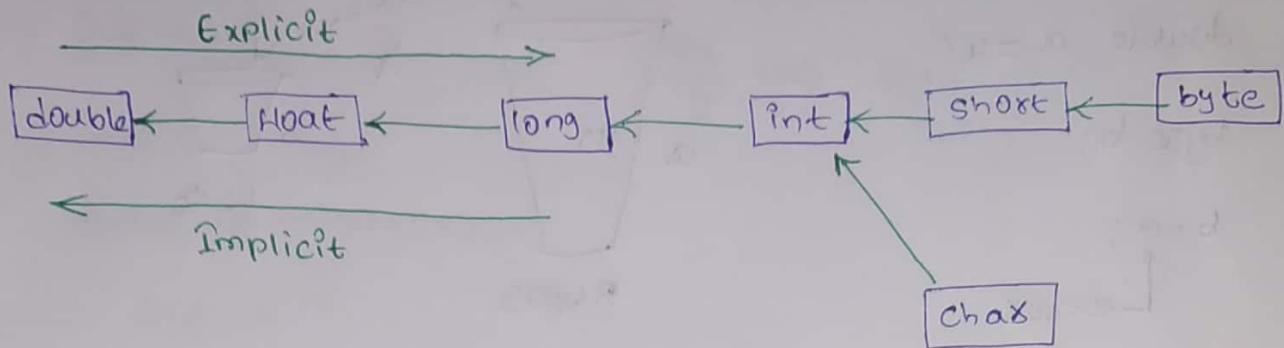
Boolean datatype (Yes/No datatype) :-

In order to handle Yes/No datatype we have boolean datatype in java and the memory allocated for a boolean type variable is JVM dependent.

List of Java's Primitive data types

Type	Size in Bytes	Range
byte	1 byte	-128 to 127
short	2 bytes	-32,768 to 32,767
int	4 bytes	-2^{31} to $+2^{31} - 1$
long	8 bytes	-2^{63} to $2^{63} - 1$
char	2 bytes	65,536 symbols
float	4 bytes	$\pm 3.40282347E + 38F$ [6-7 significant decimal digits]
double	8 bytes	$\pm 1.79769313486231570E + 308$ [15 significant decimal digits]
boolean	JVM Dependent	True False

Type casting :- It is a process of converting the data present in one data type into another data type.

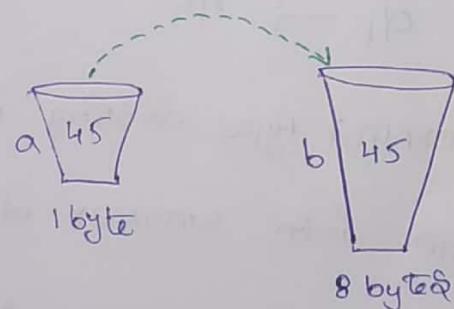


There are two types of type casting.

- i) implicit type casting
- ii) Explicit type casting

i) Implicit type casting :- It's a process of converting smaller data type into larger data type. It is automatically done by the compiler and there is no loss of data.

```
byte a = 45;  
double b;  
b = a;  
System.out.println(b);
```



O/p → 45.

ii) Explicit type casting :- It is a process of converting larger data type into a smaller data type. Compiler will not automatically perform this conversion but it would generate

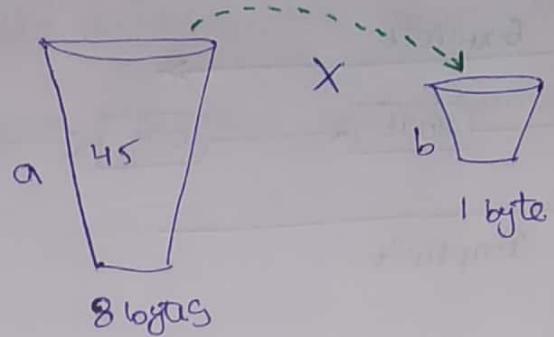
an error.

```
double a = 45;
```

```
byte b;
```

```
b = a;
```

↑ Error



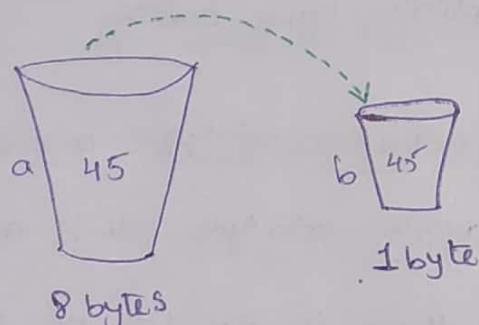
Hence, in order to resolve the shown error programmer has to explicitly provide instructions, as shown below

```
double a = 45;
```

```
byte b;
```

```
b = (byte)a;
```

```
System.out.println(b);
```



dp → 45

Hence, explicit type casting is a process of converting larger data type into smaller data type. It is explicitly performed by the programmer and there can be a loss of data.

Truncation :-

In Java, when an integer is divided by another integer the output is also an integer. This is only called as Truncation. There is loss of fractional part we can also say that the fractional part

is truncated (or) rounded off to zero.

int a = 25;

int b = 2;

int c = a/b;

System.out.println(c);

O/P → 12

$$\frac{25}{2} \Rightarrow 12 \boxed{.5} \text{ (is truncated)}$$

is truncated (or) rounded off to zero.

int a = 25;

int b = 2;

int c = a/b;

System.out.println(c);

O/P → 12

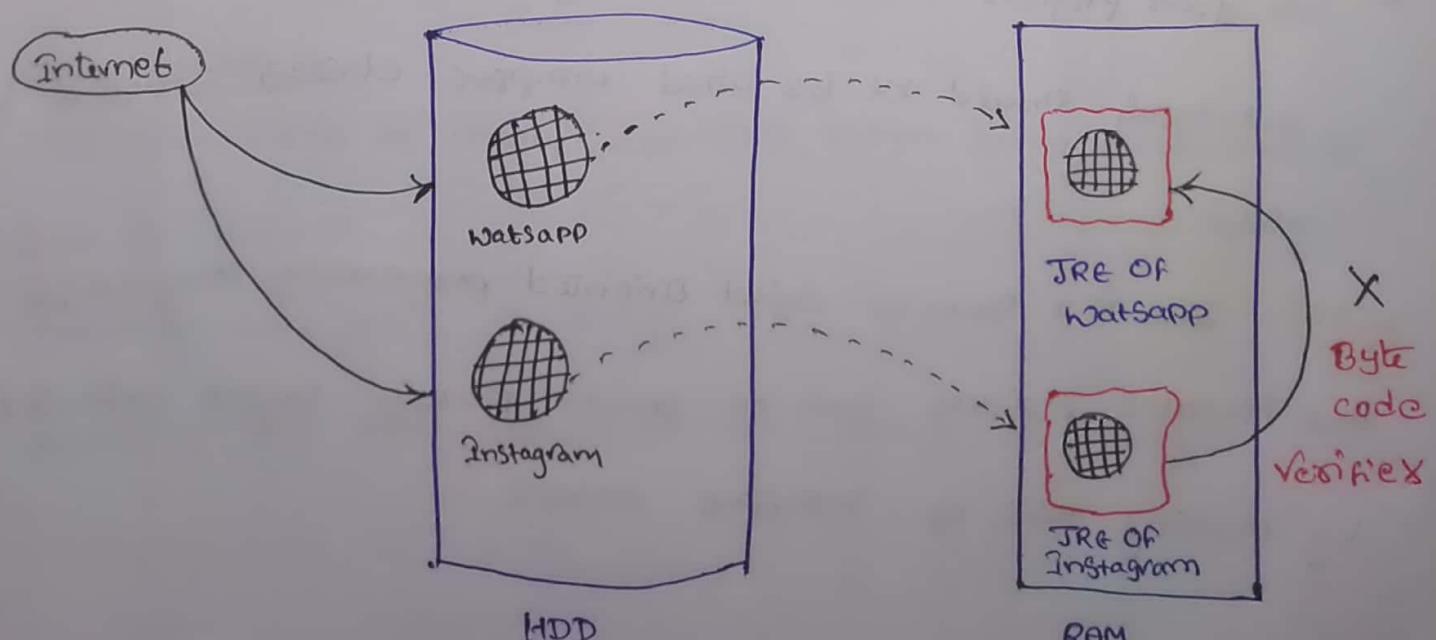
$$\frac{25}{2} \Rightarrow 12 \frac{5}{12} \quad (\text{is truncated})$$

JRE :- [Java Runtime Environment]

JRE is a region created on the RAM for the successful execution of a java program.

Note :-

Every program (or) a java application will have its own JRE regions. Illegal access between the JRE regions must be avoided for security purposes. Hence, a software called 'byte code verifier' is used prevent the illegal access between the JRE regions.



Wrapper Classes:-

Data types

```

byte a = 45;
short b = 90;
int c = 100;
long d = 150;
float e = 35.5f;
double f = 35.9;
char g = 'A';
boolean h = true;

```

Wrapper classes

```

a1
Byte a = new Byte(45);
b1
Short b = new Short(90);
Integer c = new Integer(100);
long d = new Long(150);
Float e = new Float(35.5f);
Double f = new Double(35.9);
Character g = new Character('A');
Boolean h = new Boolean(true);

```

- * Wrapper classes are inbuilt classes which are used to store the primitive data in the form of objects. Wrapper classes are used to make a project purely object oriented.
- * If your project demands speed using datatypes is a suitable solution because datatypes don't consume more time.
- * If your projects demand 100% object orientation then datatypes should not be used wrapper classes should be used.

Note:- Java is a impure object Oriented programming language
 But, however projects can be made purely object oriented by making use of wrapper classes.

Array :-

Array is a large collection of homogeneous data. Arrays are objects in java. There are two types of arrays.

- i) Regular Array
- ii) Tagged Array

Variable approach / Traditional approach :- when large amounts of data is involved in such cases variable approach is not a suitable solution because it has two major disadvantages.

- creation is difficult.
- Even if multiple variables are created and data is stored accessing the stored data becomes difficult. Hence, when large amount of data is involved variable approach should not be followed.

```
int a,b,c,d,e,f,g,h,i,j,  
int k,l,m,n,o,p,q,r,s,t;  
int u,v,w,x,y,z,aa,ab,ac,ad;
```

Array approach :

The advantage of array approach, when large amounts of data is involved.

- * Creating an array is easy.
- * Accessing the data stored in an array is easy.

```
int [] a = new int [30];
```

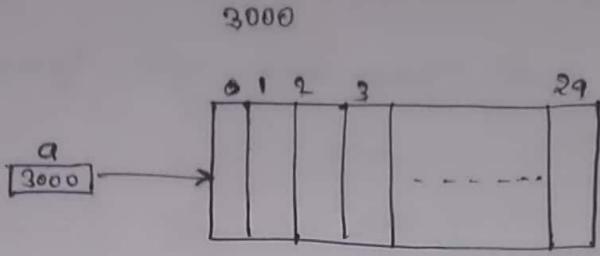
```
a[4] = 99;
```

```
a[10] = 85;
```

```
a[12] = 80;
```

⋮
⋮

```
a[29] = 28;
```



1D. Array

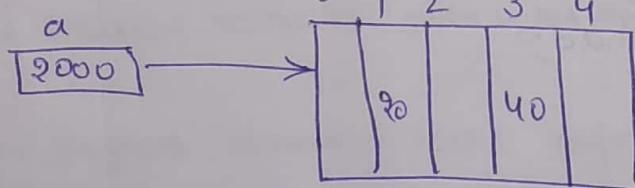
Scenario-I

To store the marks of 5 students

creation

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

Memory map



```
a[3] = 40;
```

```
a[1] = 20;
```

Scenario-II

2D Array

handle marks of 2 classes each with 5 students.

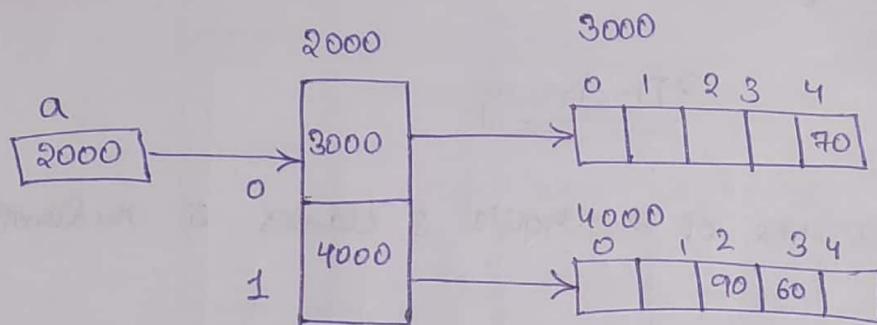
Analysis

Class	Students
0	0-4
1	0-4

creation

`int [][]a = new int [2][5];`

Memory map



`a[1][3]=60;`

`a[0][4]=70;`

`a[1][2]=90;`

Scenario

Eg: handle marks of 3 classes 4 students.

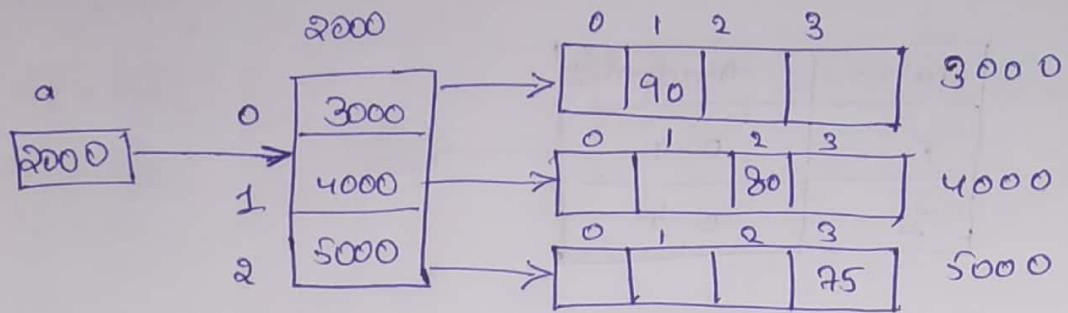
Analysis

Class	Students
0	0-3
1	0-3
2	0-3

Creation

`int [][][]a = new int [3][4];`

Memory map



$$a[0][1] = 90$$

$$a[1][2] = 80$$

$$a[2][3] = 75$$

3D-Array

Scenario III

Handle marks of 2 schools 3 classes 5 students.

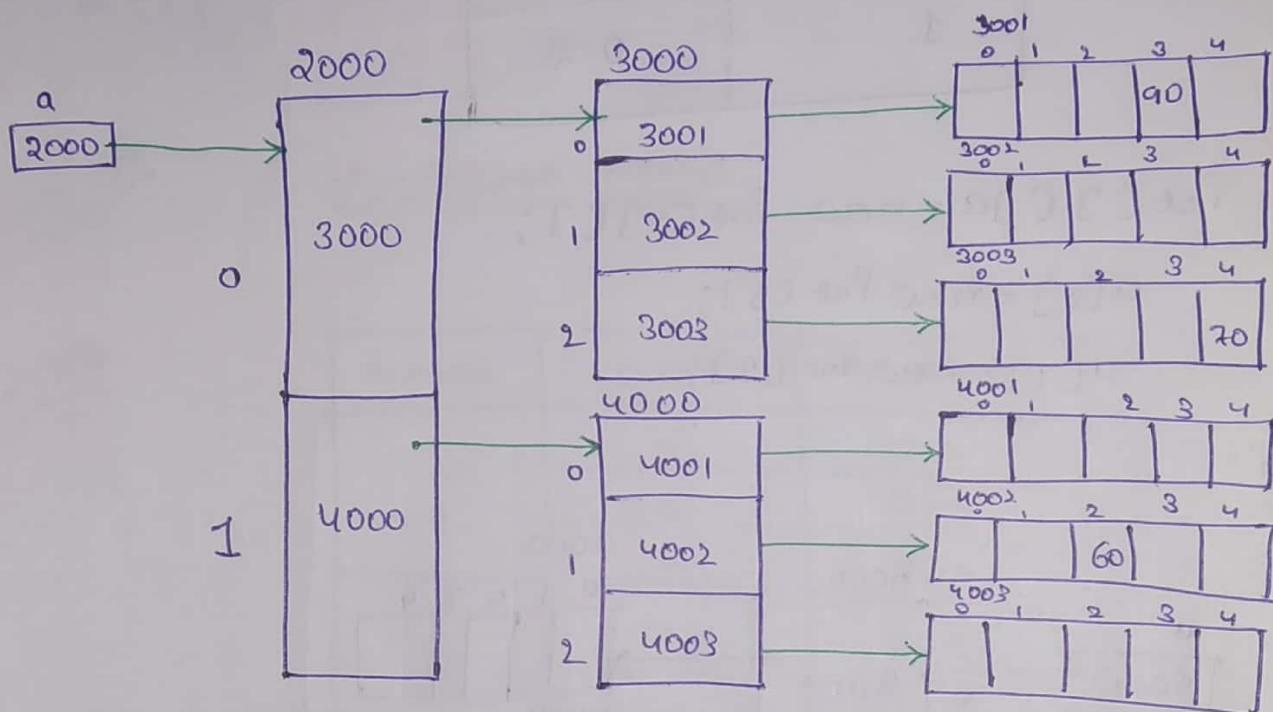
Analysis

Schools	Classes	Students
0	0	0-4
	1	0-4
	2	0-4
1	0	0-4
	1	0-4
	2	0-4

Creation :-

`int [] [] [] a = new int [2] [3] [5];`

Memory map :-



$$a[0][0][3] = 90;$$

$$a[0][2][4] = 70;$$

$$a[1][1][2] = 60;$$

Jagged Array :-

In the real world we have irregular data (or) Jagged data

Hence, In order to handle such irregular data and utilize the memory efficiently we make use of Jagged arrays instead of regular arrays.

Scenario-IV

2D-Jagged Array

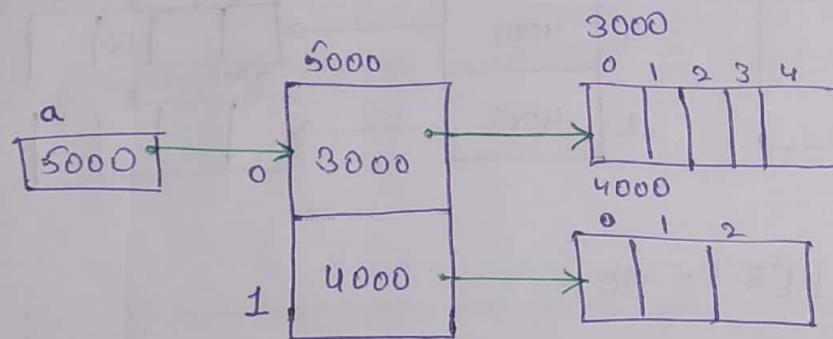
Analysis

Classes	Students
0	0-4
1	0-2

Creation:

```
int [][]a = new int [2][];
a[0] = new int [5];
a[1] = new int [3];
```

Memory Map:



Analysis:

Classes	Student
0	0-6
1	0-8
2	0-5

Creation

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

```
a[0] = new int[7];
```

```
a[1] = new int[9];
```

```
a[2] = new int[6];
```

Scenario - A

3D - Tagged Array

doubt 8/13

Schools	classes	Students
0	0	0-4
	1	0-3
	2	0-7
1	0	0-5
	1	0-1

Creation

```
int[][][] a = new int[2][ ] [ ] ;
```

```
a[0][3][ ] = new int[3][ ] ;
```

```
a[1][2][ ] = new int[2][ ] ;
```

```
a[0][0] = new int[5];
```

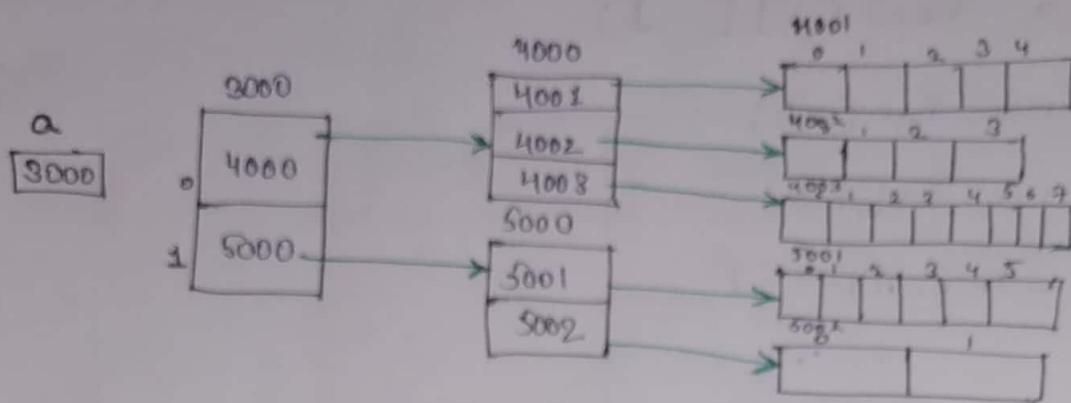
```
a[0][1] = new int[4];
```

```
a[0][2] = new int[8];
```

```
a[1][0] = new int[6];
```

```
a[1][1] = new int[2];
```

Memory Map



Scenario

3D - Jagged Array

Analysis

School#	classes	Student#
0	0	0-2
	1	0-4
	2	0-3
1	0	0-5
	1	0-7
	2	0-6
	3	0-1
2	0	0-4
	1	0-2

Creation

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

```
a[0] = new int[3][],
```

```
a[1] = new int[4][],
```

```
a[2] = new int[2][],
```

```
a[0][0] = new int[3],
```

```
a[0][1] = new int[5],
```

```
a[0][2] = new int[4],
```

`a[1][0] = new int [6];`

`a[1][1] = new int [8];`

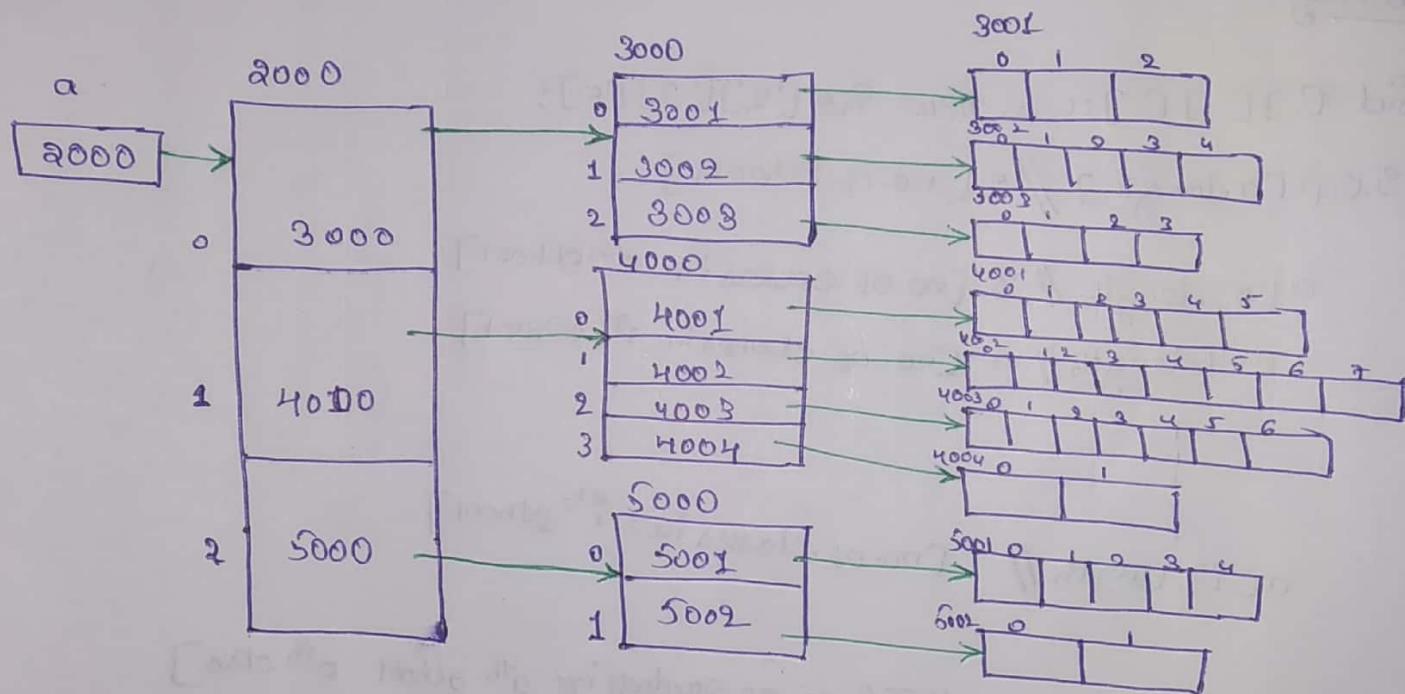
`a[1][2] = new int [7];`

`a[1][3] = new int [2];`

`a[2][0] = new int [5];`

`a[2][1] = new int [3];`

Memory Map:



Length Variable in arrays :-

1D Array:

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

`S.O. P(a.length);`

O/p:- 5 [no. of Students]

2D Array

```
int C ][ ]a = new int [3][5];
S.O.P (a.length); // 3 [no. of classes]
```

a[0].length // 5 [no. of students in 0th class]

a[1].length // 5 [no. of students in 1st class]

a[2].length // 5 [no. of students in 2nd class]

|

|

a[i].length // [no. of students in ith class]

3D Array

```
int C ][ ][ ]a = new int [2][3][5];
S.O.P (a.length); // 2 [no. of schools]
```

a[0].length // 3 [no. of classes in 0th school]

a[1].length // 3 [no. of classes in 1st school]

a[i].length // [no. of classes in ith school]

a[0][0].length // 5 [no. of students in 0th school 0th class]

a[1][2].length // 5 [no. of students in 1st school 2nd class]

a[p][j].length // [no. of students in ith school and jth class]

Scenario I

Store marks of 5 students

import java.util.Scanner;

Class Launch

{

public static void main (String args[])

{

int []a = new int [5];

Scanner Scann = new Scanner (System.in);

for (int i=0; i<5; i++)

{

s.o.p ("Enter the marks of Student " + i);

a[i] = Scann.nextInt();

}

s.o.p ("The marks of all Students are ");

for (int i=0; i<5; i++)

{

s.o.p (a[i]);

}

}

Scenario II

Store marks of 2 classes 5 students.

import java.util.Scanner;

Class Launch

{ public static void main (String args[])

{

int [][]a = new int [2][5];

Scanner Scann = new Scanner (System.in);

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

{

for (int j=0; j<a[i].length; j++)

```
{  
System.out.println("Enter the marks of " + student[i]);
```

```
a[i][j] = scan.nextInt();
```

```
}
```

```
}
```

```
s.o.p("The marks of all students are ");
```

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

```
{
```

```
for (int j=0; j<a[i].length; j++)
```

```
{
```

```
s.o.p(a[i][j]);
```

```
}
```

```
{
```

```
{
```

Scanner Store marks of 2 schools 3 classes 5 students.

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

Class Launch

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

```
{
```

```
int a[][][] = new int [2][3][5];
```

```
Scanner scan = new Scanner (System.in);
```

```
for (int i=0; i<a.length-1; i++)
```

```
{
```

```
for (int j=0; j<a[i].length-1; j++)
```

```
{
```

```
for (int k=0; k<a[i][j].length-1; k++)
```

```
{
```

```
System.out.println("Enter the marks of School " + i + " " + class[j] + " " + student[k]);
```

```

        a[i][j][k] = scan.nextInt();
    }
}

S.O.P ("The marks of the all students are");
for (int i=0; i<a.length-1; i++)
{
    for (int j=0; j<=a[i].length-1; j++)
    {
        for (int k=0; k<=a[i][j].length-1; k++)
        {
            S.O.P (a[i][j][k]);
        }
    }
}

```

Scenario IV

2D-Jagged array

class	students
0	0-4
1	0-2

```

import java.util.Scanner;

class Launch
{
    public static void main (String args[])
    {
        int[][] a = new int[2][];
        a[0] = new int[5];
        a[1] = new int[3];
        Scanner scan = new Scanner (System.in);

        for (int i=0; i<=a.length-1; i++)
        {
            for (int j=0; j<=a[i].length-1; j++)
            {

```

{

S.O.P (" Enter the marks of class "+i+" Student "+j+");

a[i][j] = scan.nextInt();

{

S.O.P (" The marks of all students are ");

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

{

for (int j=0; j<a[i].length-1; j++)

{

S.O.P (C[i][j]);

}

{

3D - Jagged array

Scenario - II

Schools	Classes	Students
0	0	0-4
0	1	0-3
0	2	0-7
1	0	0-5
1	1	0-1

```

import java.util.Scanner;
class Launch
{
    public static void main (String args[])
    {
        int[][][] a = new int [2] [ ] [ ];
        a [0] = new int [3] [ ];
        a [1] = new int [2] [ ];
        a [0] [0] = new int [5];
        a [0] [1] = new int [4];
        a [0] [2] = new int [8];
        a [1] [0] = new int [6];
        a [1] [1] = new int [2];
        Scanner scan = new Scanner (System.in);
        for (int i = 0; i <= a.length - 1; i++)
        {
            for (int j = 0; j <= a[i].length - 1; j++)
            {
                for (int k = 0; k <= a[i] [j].length - 1; k++)
                {
                    System.out.println ("Enter the marks of school " + i + " class");
                    a [i] [j] [k] = scan.nextInt ();
                }
            }
        }
        System.out.println ("The marks of all students are ");
    }
}

```

```

for (int i=0; i<=a.length-1; i++)
{
    for (int j=0; j<=a[i].length-1; j++)
    {
        for (int k=0; k<=a[i][j].length-1; k++)
        {
            System.out.print(a[i][j][k]);
        }
    }
}

```

Different types of Declaration of Arrays

1D

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

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

```
int a[] = new int [ ] { 90, 80, 70, 60, 50 };
```

↑
never mention size

```
int a[] = new { 90, 80, 70, 60, 50 };
```

2D

```
int [] [ ] a = new int [2] [5];
```

```
int [ ] a [ ] = new int [2] [5];
```

```
int a [ ] [ ] = new int [2] [5];
```

```
int a [ ] [ ] = new int [2] [5] { { 10, 20, 30, 40, 50 }, { 60, 70, 80, 90, 100 } };
```

↑
never mention size

```
int a [ ] [ ] = new int { { 10, 20, 30, 40, 50 }, { 60, 70, 80, 90, 100 } };
```

3D

```
int [ ] [ ] [ ] a = new int [2] [3] [5];
```

```
int [ ] [ ] a [ ] = new int [2] [3] [5];
```

```
int [ ] [ ] a [ ] = new int [2] [3] [5];
```

```
int a [ ] [ ] [ ] = new int [2] [3] [5];
```

```
int a [ ] [ ] [ ] = new int [2] [3] [5] { { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9, 10 }, { 11, 12, 13, 14, 15 } }, { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9, 10 }, { 11, 12, 13, 14, 15 } } };
```

↑
Never mention size

```
int a [ ] [ ] [ ] = new int { { { 1, 1, 1, 1, 1 }, { 2, 2, 2, 2, 2 }, { 3, 3, 3, 3, 3 } }, { { 1, 1, 1, 1, 1 }, { 2, 2, 2, 2, 2 }, { 3, 3, 3, 3, 3 } } };
```

Disadvantages of Arrays :-

* 1. Array can store only homogeneous data.

Heterogeneous data cannot be stored.

Ex:

Class Launch

{

 Public static void main (String args [])

{

 Int [] a = new int [5];

 a[0] = 1;

 a[1] = 'A'

 a[2] = true; // Error

 a[3] = 35.5f; // Error

 a[4] = "RAM"; // Error

}

Disadvantage - 2

→ The size of the array is fixed it is not going to

dynamically grow or shrink in size during execution.

Ex:

Class Launch

{

 Psvm (String args [])

{

 Int a[] = new int [3];

 a[0] = 100;

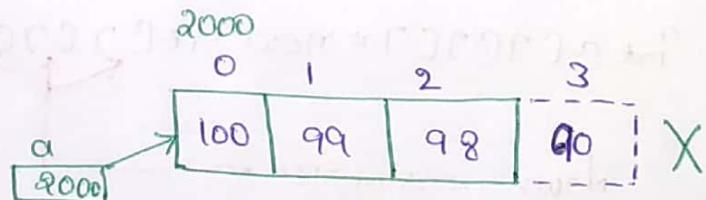
 a[1] = 99;

 a[2] = 98;

 a[3] = 90; // Error

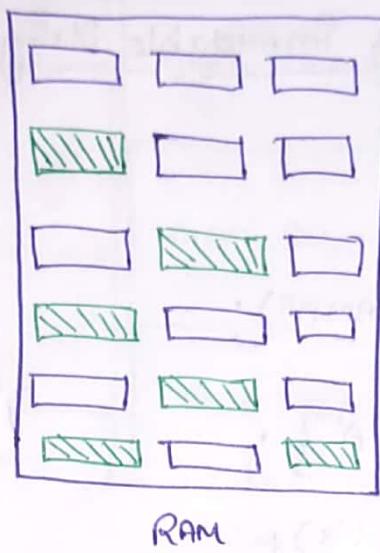
}

{

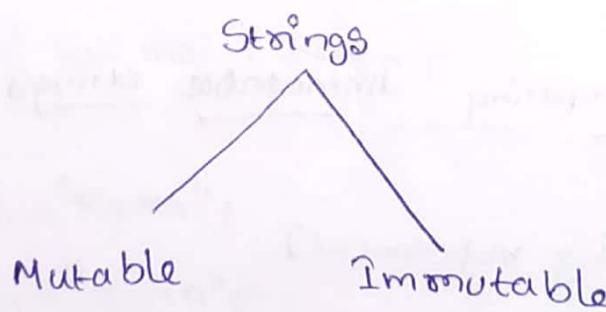


Disadvantage - 3

→ Array is demand contiguous memory location. If the memory spaces are dispersed array will not be able to utilise dispersed memory locations.



String :-



Eg:- Email
Password

Eg:- Name
Gender

[String Buffer]

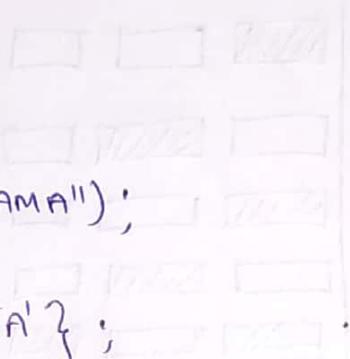
[String]

[String Builder]

* String is a collection of characters enclosed within the double quotes code. In java strings are objects and they are not null terminated.

Different ways of creating immutable String :-

1. String s1 = "RAMA";



2. String s2 = new String ("RAMA");

3. char a[] = {'R','A','M','A'};

String s3 = new String(a);

4. Char [] a = {'R','A','M','A'};

String s4 = new String (a);

Different ways of comparing Immutable strings :-

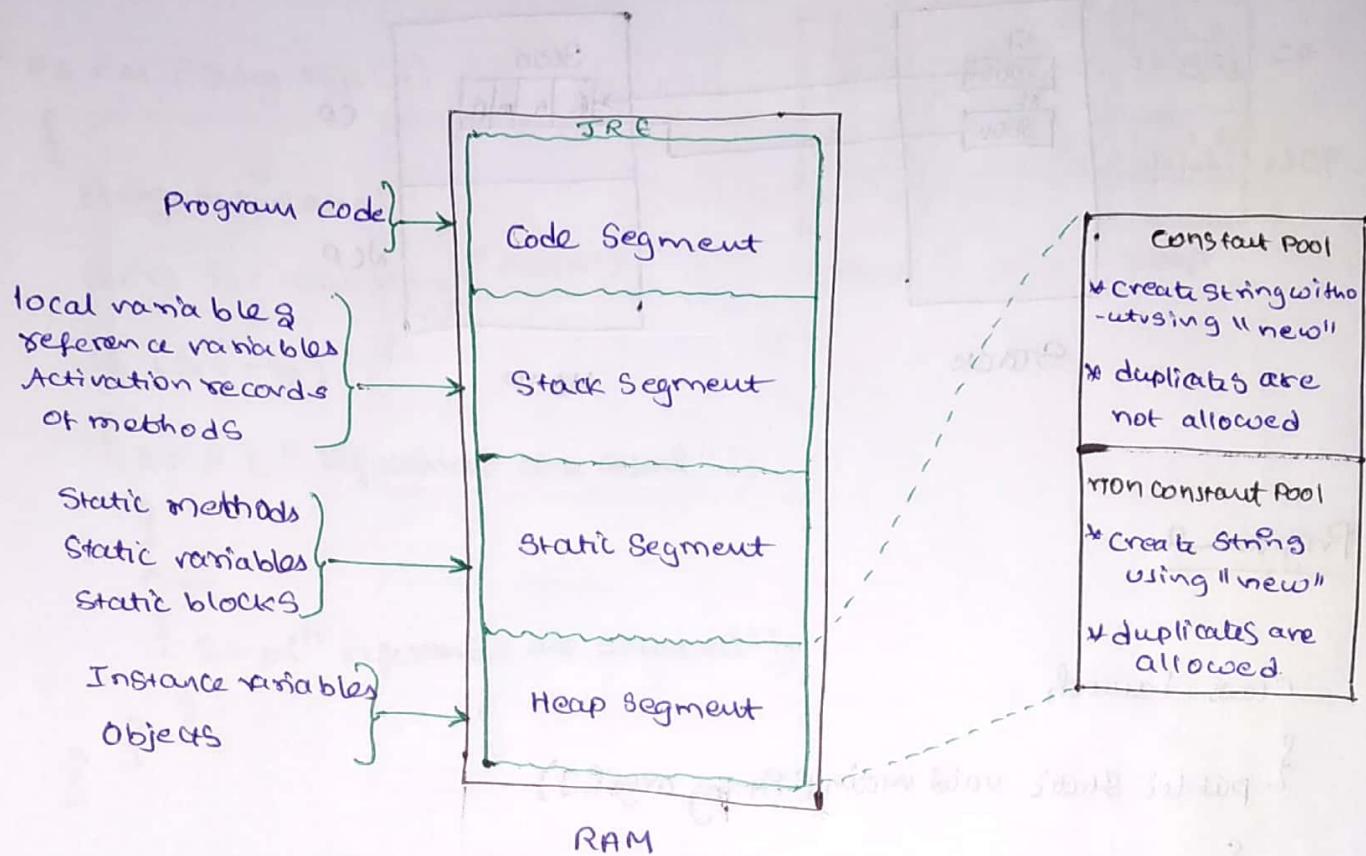
1. s1 == s2 [Comparing references]

2. s1.equals(s2) [Comparing Values]

3. s1.compareTo(s2) [values are compared character by character]

4. s1.equalsIgnoreCase(s2) [values are compared Ignoring case]

Different segments of JRE :-



Program - 1

Class Launch

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

```
String s1 = "RAMA";
```

```
String s2 = "RAMA";
```

```
if (s1 == s2)
```

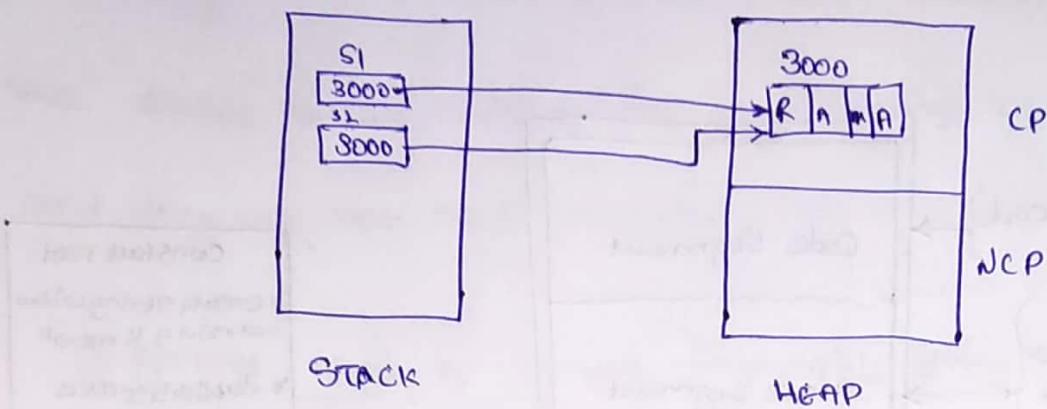
```
{  
    System.out.println("references are equal");  
}
```

```
else
```

```
{  
    System.out.println("references are unequal");  
}
```

```
}
```

O/p :- strings are references are equal.



Program-2

Class Launch

```
{ public static void main(String args[])
{
    String s1 = "RAMA";
    String s2 = "RAMA";
    if (s1.equals(s2) == true)
    {
        System.out.println("Strings are equal");
    }
    else
    {
        System.out.println("Strings are unequal");
    }
}
```

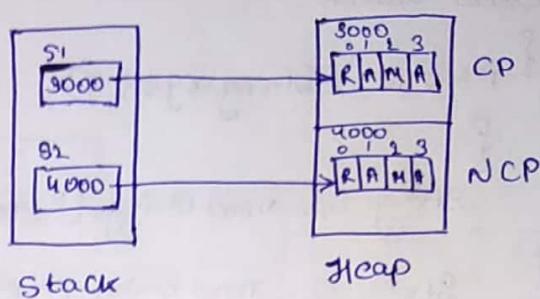
O/p :- Strings are equal.

Program - 3

Class Launch

```

{
    public void main (String args[])
    {
        String s1 = "RAMA";
        String s2 = new String ("RAMA");
        if (s1 == s2)
        {
            System.out.println ("References are equal");
        }
        else
        {
            System.out.println ("References are unequal");
        }
    }
}
  
```



O/P :- References are unequal.

Program - 4

Class Launch

```

{
    public void main (String args[])
    {
        String s1 = "RAMA";
        String s2 = new String ("RAMA");
        if (s1.equals (s2) == true)
        {
            System.out.println ("Strings are equal");
        }
        else
        {
            System.out.println ("Strings are unequal");
        }
    }
}
  
```

O/P :- Strings are equal.

Program-5

class Launch

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

```
String s1 = new String ("RAMA");
```

```
String s2 = new String ("RAMA");
```

```
if (s1 == s2)
```

```
{ System.out.println ("references are equal"); }
```

```
}
```

```
else
```

```
{ System.out.println ("references are un-equal"); }
```

```
}
```

```
}
```

O/P :- references are unequal.

Program-6

class Launch

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

```
String s1 = new String ("RAMA");
```

```
String s2 = new String ("RAMA");
```

```
if (s1.equals (s2) == true)
```

```
{ System.out.println ("Strings are equal"); }
```

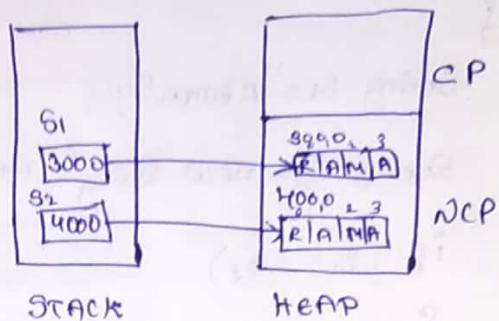
```
}
```

```
else
```

```
{ System.out.println ("Strings are un-equal"); }
```

```
}
```

O/P :- strings are equal.



Program - 7

```

class Launch
{
    public static void main (String args[])
    {
        String s1 = "RAMA";
        String s2 = "SITA";
        String s3 = "RAMA" + "SITA";
        String s4 = "RAMA" + "SITA";

        if (s3 == s4)
        {
            System.out.println("references are equal");
        }
        else
        {
            System.out.println("references are unequal");
        }
    }
}

```

O/p :- references are equal.

Program - 8

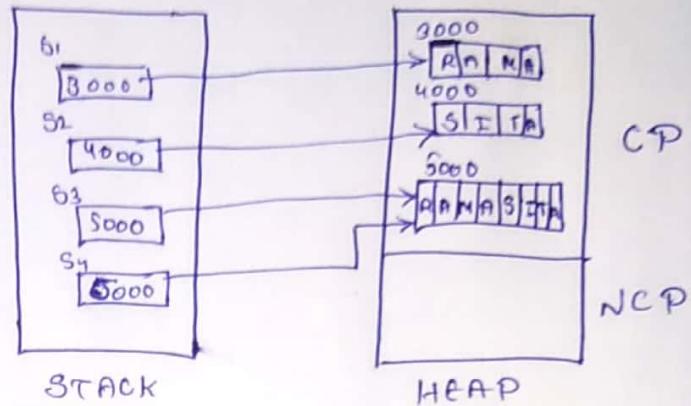
```

class Launch
{
    public static void main (String args[])
    {
        String s1 = "RAMA";
        String s2 = "SITA";
        String s3 = "RAMA" + "SITA";
        String s4 = "RAMA" + "SITA";

        if (s3.equals(s4) == true)
        {
            System.out.println("Strings are equal");
        }
        else
        {
            System.out.println("Strings are unequal");
        }
    }
}

```

O/p :- strings are equal.



Program - 9 :-

Class Launch

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

```
    String s1 = "RAMA";
    String s2 = "SITA";
    String s3 = s1 + s2;
    String s4 = s1 + s2;
```

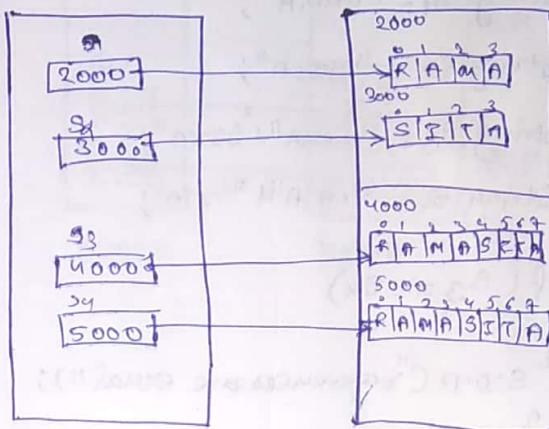
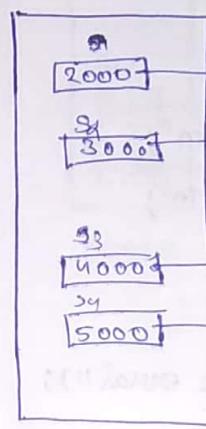
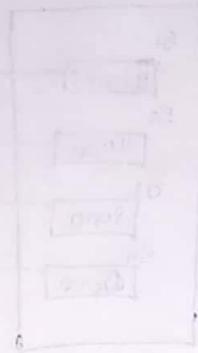
```
    if (s3 == s4)
```

```
    { S.O.P ("references are equal");
    }
```

```
    else
```

```
    { S.O.P ("references are un-equal");
    }
```

? O/P :- references are un-equal.



CP

NCP

STACK

HEAP

Program - 10 :-

Class Launch

```
{ P.S.VM C string args[])
{
```

```
    String s1 = "RAMA";
    String s2 = "SITA";
    String s3 = s1 + s2;
    String s4 = s1 + s2;
```

```
    if (s1.equals(s2) == true)
    { S.O.P ("strings are equal");
    }
```

```
    else
    { S.O.P ("strings are un-equal");
    }
```

? O/P:- strings are equal.

Program - 11

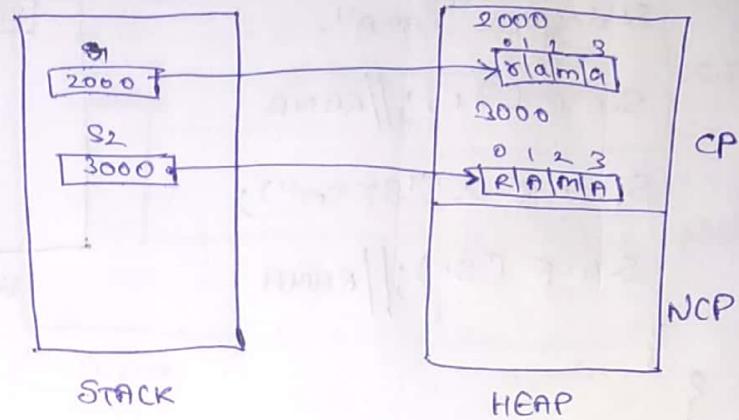
Class Launch

```

{
    public static void main (String args[])
    {
        String S1 = "xama";
        String S2 = "RAMA";
        if (S1.equals(S2) == true)
        {
            S.O.P ("Strings are equal");
        }
        else
        {
            S.O.P ("Strings are un-equal");
        }
    }
}

```

O/p :- strings are un-equal.



Program - 12

Class Launch

```

{
    public static void main (String args[])
    {
        String S1 = "xama";
        String S2 = "RAMA";
        if (S1.equalsIgnoreCase(S2))
        {
            S.O.P ("Strings are equal");
        }
        else
        {
            S.O.P ("Strings are un-equal");
        }
    }
}

```

O/P :- strings are equal.

Program - 13

Class Launch

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

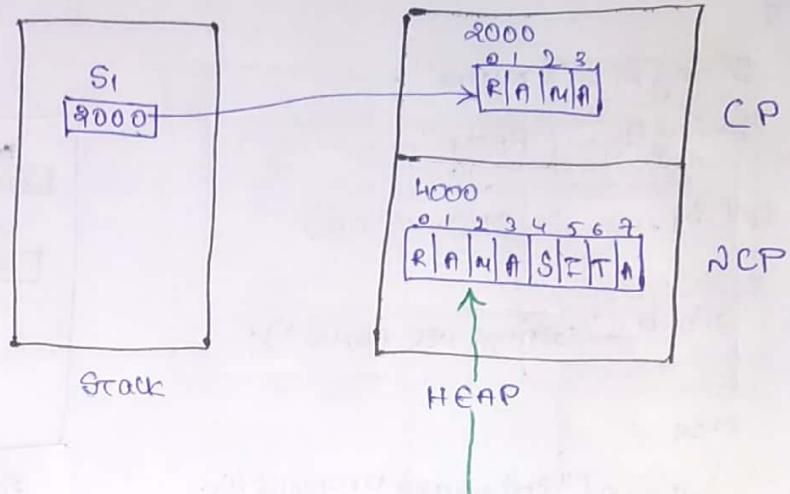
String s1 = "RAMA";

s.o.p(s1); //RAMA

s1.concat("SITA");

s.o.p(s1); //RAMA

}



No reference is pointing
at this object

Program - 14

Class Launch

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

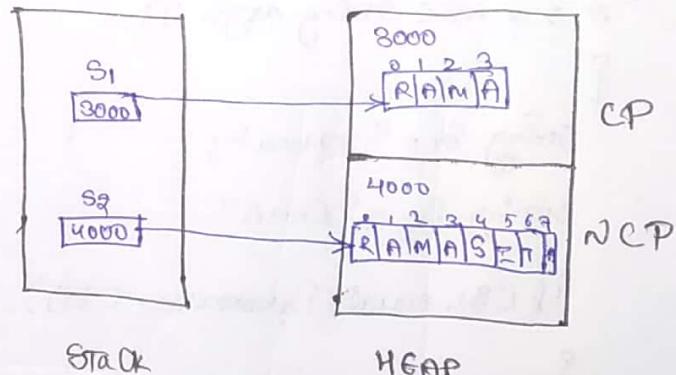
String s1 = "RAMA";

s.o.p(s1); //RAMA

String s2 = s1.concat("SITA");

s.o.p(s2); //RAMASITA

}



Program - 15

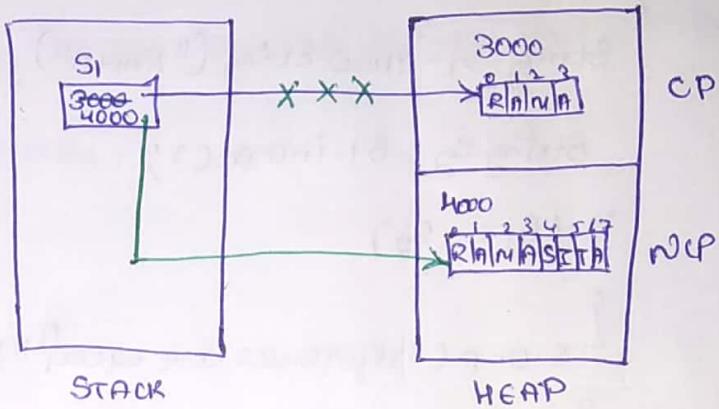
Class Launch

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

```
        String s1 = "RAMA";  
        System.out.println(s1); // RAMA
```

```
        s1 = s1.concat("SITA");  
        System.out.println(s1); // RAMASITA
```

```
} } }
```



Program - 16

Class Launch

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

```
        String s1 = new String("RAMA");
```

```
        String s2 = s1;
```

```
        if (s1 == s2)
```

```
            System.out.println("references are equal");
```

```
} }
```

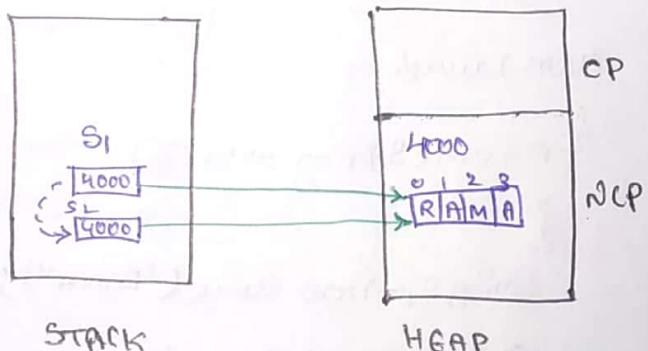
```
else
```

```
    System.out.println("references are unequal");
```

```
}
```

```
}
```

O/P :- References are equal.



Intern method

- * Intern method is used to take a copy of the object/String from the "NCP" and place it on the "CP".
But, however if the same object is already present in the Constant pool another object is not created rather the address of the existing object is returned.

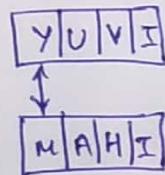
CompareTo:-

Class Launch

```

{
    public static void main(String[] args)
    {
        String s1 = "SACHIN";
        String s2 = "SAURAV";
        int a = s1.compareTo(s2);
        System.out.println(a); // -18
    }
}
  
```

- * String s1 = "YUVI";
String s2 = "NAHI";



[12 is returned]

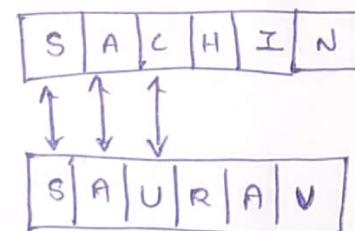
$$\begin{array}{r}
 89 \\
 77 \\
 \hline
 12
 \end{array}$$

String s1 = "SACHIN";

String s2 = "SAURAV";

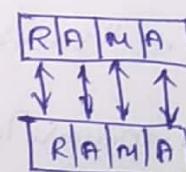
[-18 is returned]

[- shows that s2 is greater]



$$\begin{array}{r}
 83 \ 65 \ 67 \\
 83 \ 65 \ 75 \\
 \hline
 0 \ 0 \ -18
 \end{array}$$

- * String s1 = "RAMA";
String s2 = new String("RAMA");

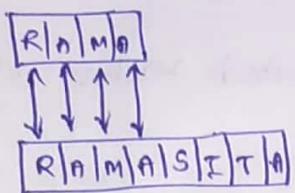


[0 is returned]

$$\begin{array}{r}
 82 \ 65 \ 72 \ 65 \\
 82 \ 65 \ 72 \ 65 \\
 \hline
 0 \ 0 \ 0
 \end{array}$$

String s₁ = "RAMA";

String s₂ = "RAMASITA";



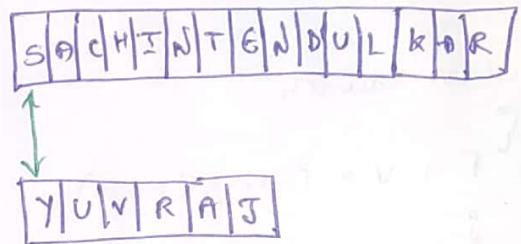
[-1 is returned]

'1' is the no. of extra characters.

'-' sign shows that s₂ is greater.

String s₁ = "SACHINTENDULKAR";

String s₂ = "YUVRAJ";



$$\begin{array}{r} 83 \\ 89 \\ \hline -6 \end{array}$$

Program-19

Class Launch

```
{  
    public static void main (String args[])  
    {  
        String s1 = "YUVRAJ";  
        String s2 = "SACHIN";  
        int temp = s1.compareTo(s2);  
  
        if (temp > 0)  
        {  
            System.out.println ("String s1 is greater than string s2");  
        }  
        else if (temp < 0)  
        {  
            System.out.println ("String s2 is greater than string s1");  
        }  
        else  
        {  
            System.out.println ("Both strings are equal");  
        }  
    }  
}
```

s.o.p("String1 is greater than String2");

}

else

{

s.o.p("Strings are equal");

}

}

O/P :- string s₁ is greater than string s₂.

Program - 20

Class Launch

{

psvm C String args[2]

{

String s₁ = "URAMA";

String s₂ = "rama";

int temp = s₁.compareToIgnoreCase(s₂);

if (temp > 0)

{

s.o.p("String1 is greater than String2");

}

else if (temp < 0)

{

s.o.p("String2 is greater than String1");

}

else

{

s.o.p("Strings are equal");

}

}

O/P :- Strings are equal.

Few Inbuilt methods of String Class

Class Launch

{

PSvm (String args[])

{

String s = "RajaRamMohanRoy";

System.out.println(s.toUpperCase()); // RAJARAMMOHANROY

s.toLowerCase(); // rajaRammohansoy

s.charAt(0); // R

s.indexOf('R'); // 0

s.lastIndexOf('R'); // 12

s.startsWith("Raja"); // True

s.startsWith("Rani"); // False

s.endsWith("Roy"); // True

s.endsWith("toy"); // False

s.substring(7, 12); // Mohan

s.substring(12, 15); // Roy

s.substring(7); // MohanRoy

s.contains("Mohan"); // True

s.contains("Rohan"); // False

Mutable Strings:-

In order to create mutable strings we have two classes in Java.

1) StringBuffer

2) String Builder

Eg:1

Class Launch

```
{ public static void main (String [] args)
{
    StringBuffer sb = new StringBuffer ();
    sb.append ("RAMA");
    System.out.println (sb); // RAMA
    sb.append ("SITA");
    System.out.println (sb); // RAMASITA
}}
```

Eg:2

Class Launch

```
{ public static void main (String [] args)
{
    String Builder sb = new String Builder ();
    sb.append ("RAMA");
    System.out.println (sb); // RAMA
    sb.append ("SITA");
    System.out.println (sb); // RAMASITA
}}
```

Eg:3

Class Launch

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

```
        StringBuider sb = new StringBuider();
```

```
        S.O.P(sb.capacity()); //16
```

```
        sb.append("SACHIN");
```

```
        S.O.P(sb.capacity()); //16
```

```
        sb.append(" HE IS A BATSMAN");
```

```
        S.O.P(sb.capacity()); //34
```

```
        sb.append(" HE IS ALSO AN NP");
```

```
        S.O.P(sb.capacity()); //70
```

```
        S.O.P(sb);
```

```
}
```

69

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	---	
S	A	C	H	I	N		H	E		I	S		A		B	A	T	S	N	A	W		H	E		I	S		A	L	S	O		A	N		M	P		---

16

$$\text{new size} = \lceil \text{old size} \times 2 \rceil + 2$$

$$[\lceil 34 \times 2 \rceil + 2]$$

$$= \lceil 68 \times 2 \rceil + 2$$

$$= 68 + 2$$

$$= 70$$

$$= 70$$

$$= 32 + 2$$

$$= 34$$

Eg:4

* String Buffer is same as String Builder *

[which is written above]

Eg: 5

Class Launch

```
{
    public static void main(String[] args)
    {
        String Builder sb = new String Builder();
        System.out.println(sb.capacity()); // 16
        sb.ensureCapacity(60);
        System.out.println(sb.capacity()); // 60
        sb.ensureCapacity(100);
        System.out.println(sb.capacity()); // 122
        sb.ensureCapacity(70);
        System.out.println(sb.capacity()); // 122
    }
}
```

Eg: 6

* String Buffer holds the same program as String Builder *

Eg: 7

Class Launch

```
{
    public static void main(String[] args)
    {
        String Buffer sb1 = new String Buffer("RAMA");
        System.out.println(sb1.capacity()); // 20
        String Buffer sb2 = new String Buffer();
        System.out.println(sb2.capacity()); // 16
        sb2.append("RAMA");
        System.out.println(sb2.capacity()); // 16
    }
}
```

Note:-

* String Builder holds the same good for String Buffer.

String Buffer

1. String Buffer is slow in execution.
2. It can't be used for multi-threading.
3. Race condition will not occur.
4. All the methods in this class are synchronized.
5. It is thread Safe.

String Builder

1. String Builder is fast in execution.
2. It can be used for multi-threading.
3. Race condition will occur.
4. All the methods in this class are not synchronised.
5. It is not thread Safe.

Note:-

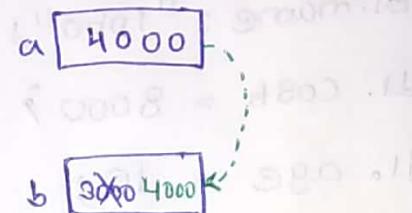
Both String Buffer and String Builder have the initial capacity of 16 locations and both are used to build mutable strings.

Value type assignment and reference type assignment.

Value type assignment :-

Class Launch

```
{  
    public static void main (String args [] )  
    {  
        int a = 4000;  
        int b = 3000;  
        System.out.println (b); // 3000  
        b = a;  
        System.out.println (b); // 4000  
    }  
}
```



Reference type assignment :-

X1 Class Dog

```
{  
    String name;  
    int cost;  
    int age;  
  
    void disp()  
    {  
        System.out.println (name);  
        System.out.println (cost);  
        System.out.println (age);  
    }  
}
```

Class Launch

{

PSVMC String args(7)

{

Dog d1 = new Dog();

d1.name = "Lobo";

d1.cost = 8000;

d1.age = 10;

d1.disp();

Dog d2;

d2 = d1;

d2.disp();

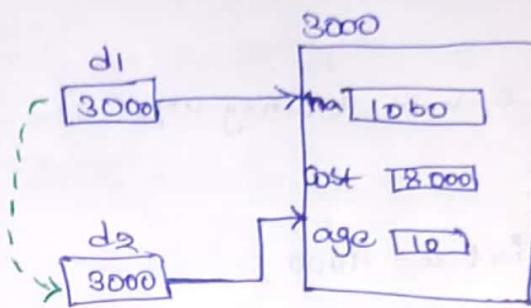
}

}

O/p :- Lobo

8000

10



*2

```
class Dog
{
    String name;
    int cost;
    int age;

    void disp()
    {
        System.out.println(name);
        System.out.println(cost);
        System.out.println(age);
    }
}
```

class Launcher

```
{ public static void main(String args[])
{
    Dog d1 = new Dog();
    d1.name = "Lobo";
    d1.cost = 8000;
    d1.age = 10;
    d1.disp();
}}
```

```
Dog d2 = new Dog();
d2.name = "Rocky";
d2.cost = 9000;
d2.age = 6;
d2.disp();
```

Dog dəg

$$d_3 = d_1,$$

ds.disp();

$$d_3 > d_2;$$

d3. disp();

3

3

O/p :-

lobo

8000

10

Rocky

9 000

6

Lobo

8000

t

ROCKY

9000

6

* 3

Class Dog

{

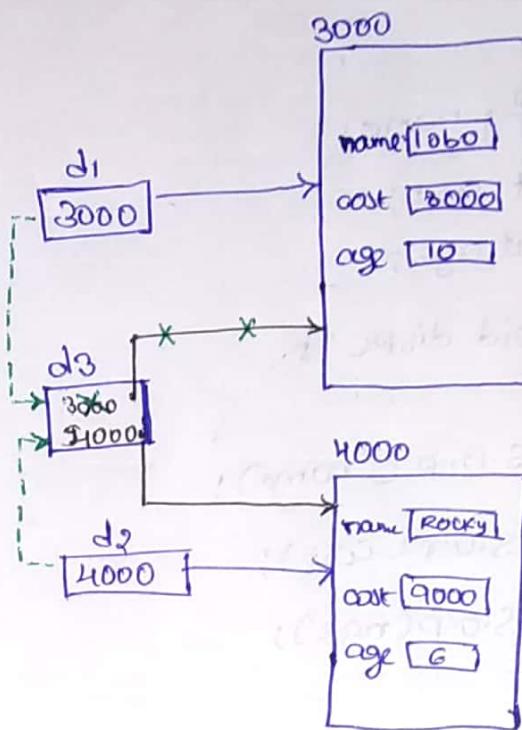
String name;

```
int cost;
```

Put age;

void disp()

{



```
S.o.p(name);  
S.o.p(cost);  
S.o.p(age);  
}  
}
```

Class Launch

```
{  
    Psvm(String args[])  
{
```

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

```
d1.name = "lolo";
```

```
d1.cost = 8000;
```

```
d1.age = 10;
```

```
d1.disp();
```

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

```
d2.name = "Rocky";
```

```
d2.cost = 9000;
```

```
d2.age = 6;
```

```
d2.disp();
```

```
Dog d3;
```

```
d3 = d1;
```

```
d3.name = "Tommy";
```

```
d3.age = 2;
```

```
d1.disp();
```

```
d2 = d3;
```

```
} d2.disp();
```

```
}
```

O/P :-

10bo

8000

10

ROCKY

9000

6

10bo

Tommy

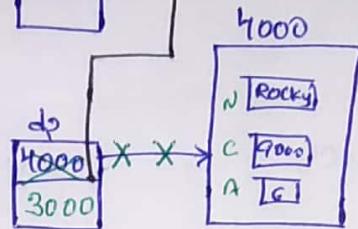
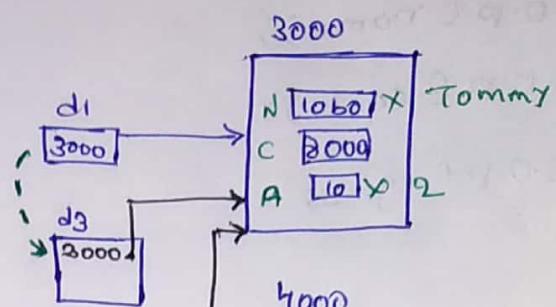
8000

2

Tommy

8000

2



- * In value type assignment the data i.e., stored inside one variable is transferred / assigned to another variable.
- * In reference type assignment the address i.e., stored inside one reference variable is transferred to another reference variable. Here, there is no transfer of data rather what gets transferred is the 'address'.

Local variable and Instance variables :-

Local variables

Instance variables

- * They are created inside a method.
- * Memory is allocated in stack segment.
- * Memory is de-allocated when the method terminates./ when the activation record of the method is deleted.
- * initialised by programmer.
- * Can't be used without initialization.
- * Irrespective of the access modifiers used, local variables are accessible only inside the method where they are created.

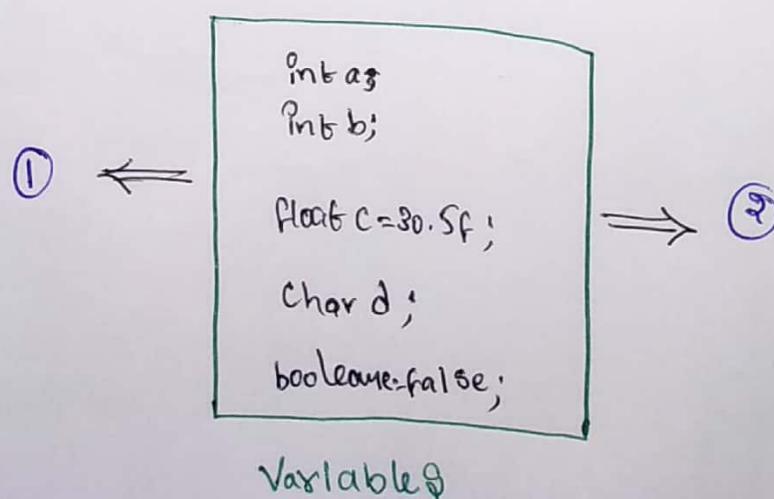
- * They are created directly inside a class.

- * Memory is allocated in heap segment.
- * Memory is de-allocated by garbage collector.

- * initialised by JVM.

- * Can be used without initialization.

- * Irrespective of the access modifiers used, variables are accessible throughout the class.



①

Class Launch

```
{
    int a;
    int b;
    float c;
    char d;
    boolean e;
}
```

* Instance variables *

• Data members of class

② Main Method

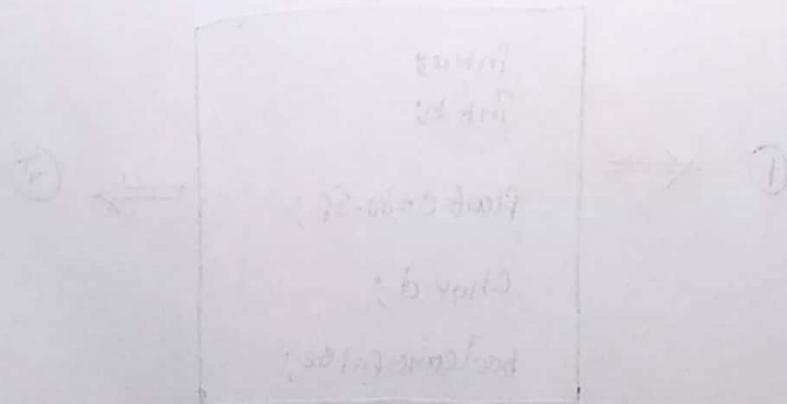
Class Launch

```
{
    public static void main(String args[])
    {
        int a = 10;
        int b = 20;
        float c = 30.5f;
        char d = 'A';
        boolean e = false;
    }
}
```

* Local variables *

• Local variable are created

• Local variable are created at declaration



①

```

Class Launch
{
    int a;
    int b;
    float c;
    char d;
    boolean e;
}

```

②

```

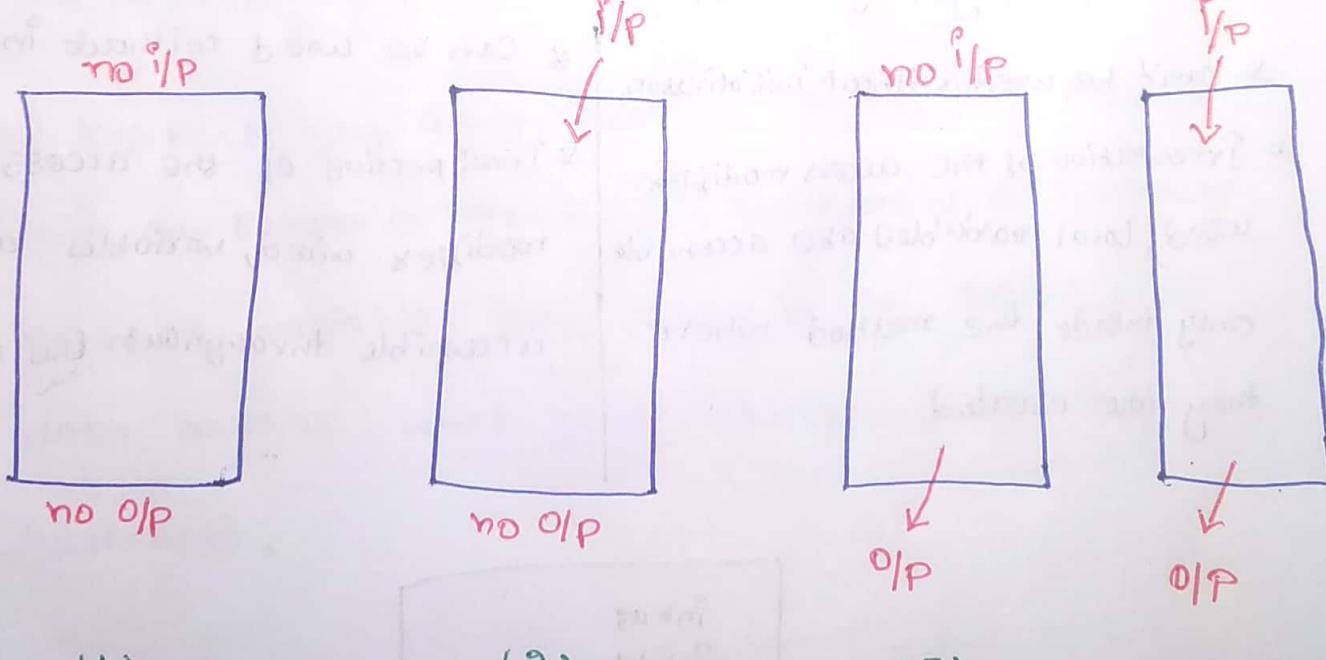
Class Launch
{
    public static void main(String args[])
    {
        int a=10;
        int b=20;
        float c=30.5f;
        char d='A';
        boolean e=false;
    }
}

```

* Instance variables *

* Local variables *

Types of methods [with respect to Input and Output]



*

class calculator

{

int a;

int b;

void disp()

{

a=10;

b=20;

int c=a+b;

s.o. P(c);

}

}

Class Launch

{

public static void main (String args[])

{

calculator calc = new calculator();

calc.disp();

}

}

(b)

No i/p - no op

METHOD-1

Class Calculator

```
{ int a;
int b;
void disp()
{
    a=10;
    b=20;
    int c=a+b;
    S.O.P (c);
}
```

PSVM (String args[]);

Calculator calc = new Calculator();

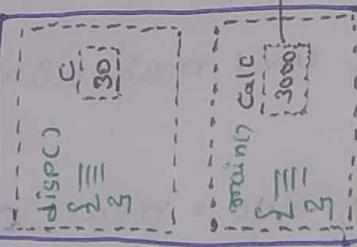
calc. disp();

O/P

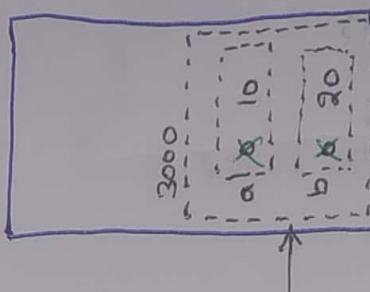
30

CODE

Activation record of disp()



Activation record of main()



STACK

HEAP
(Instance variables)

STATIC

Method -2

Tipe dan no olo

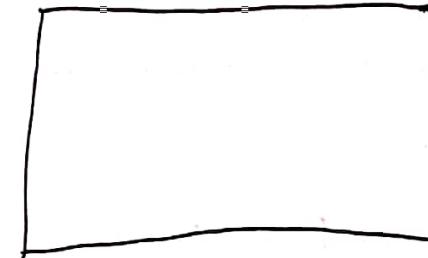
```

class Calculator
{
    int a;
    int b;
    void disp (int x, int y)
    {
        a=x;
        b=y;
        int c = a+b;
        System.out.println(c);
    }
}
class Launch
{
    public static void main (String args[])
    {
        Calculator calc = new Calculator();
        calc.disp(100, 200);
    }
}

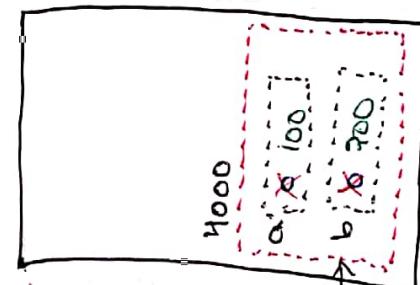
```



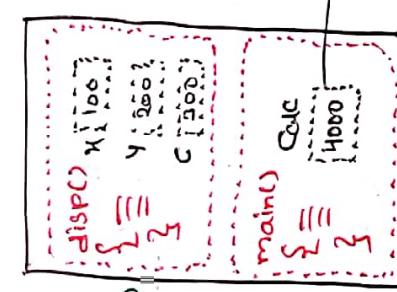
O/P



STATE



Неск



ACTIVITIES

Record of
meeting

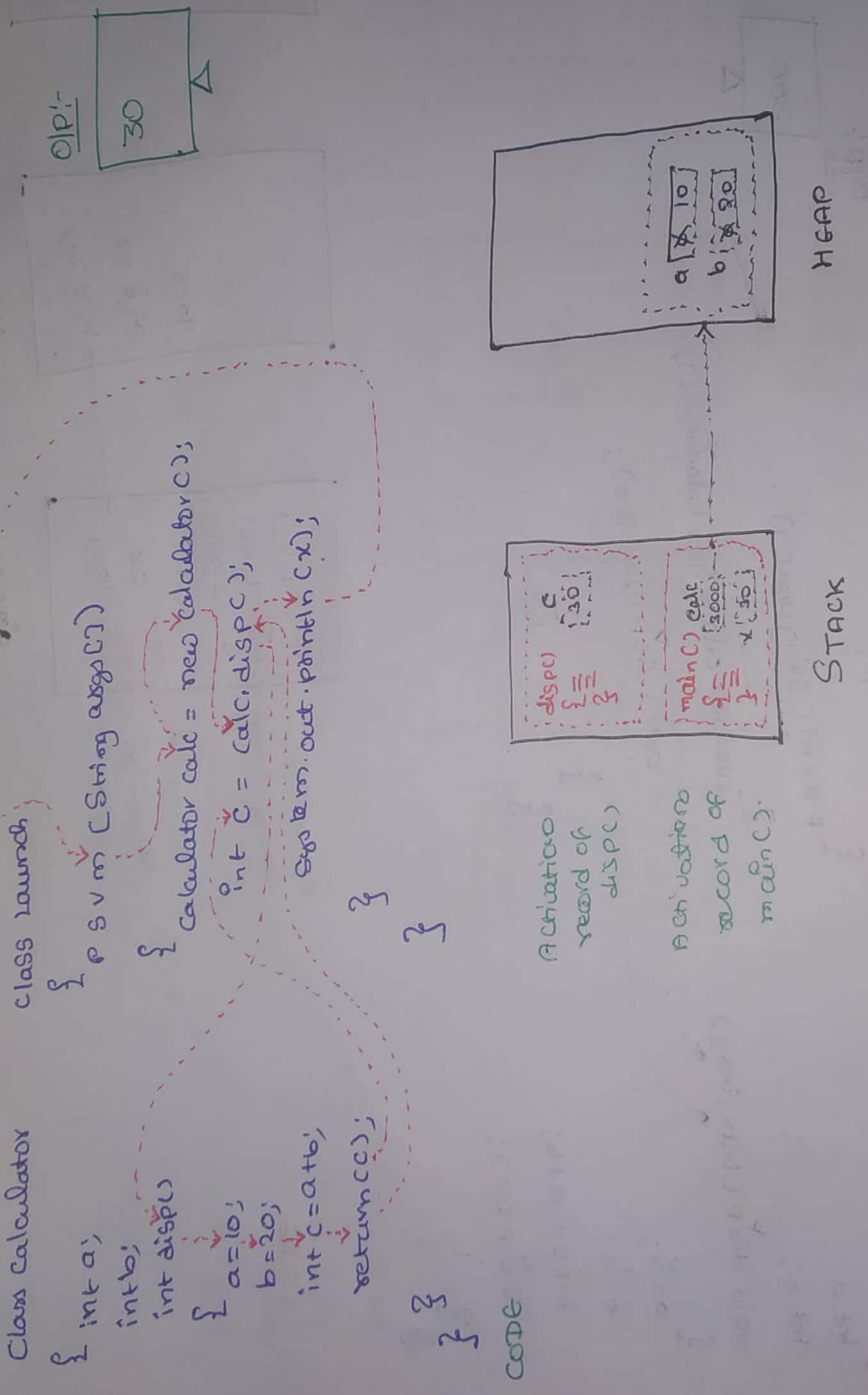
۱۰

STACK

Type - 3 No Impact & Output

```

graph TD
    Start[Class calculator] --> DeclA[int a;]
    Start --> DeclB[int b;]
    DeclA --> Add[c = a + b;]
    DeclB --> Add
    Add --> Output[cout << c;]
    Add --> ValC[c = 10;]
    Add --> ValB[b = 20;]
    Output --- Return[return c;]
    ValC --- Return
    ValB --- Return
    Output --- Return
  
```



OS Type-4 :- TIP & OLP

Class Calculator

```

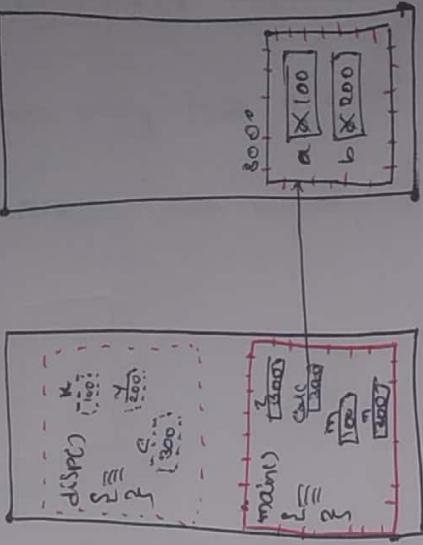
{ int a;
int b;
int disp (int x, int y)
{
    int m = 100;
    int n = 200;
    int z = calc. disp(x,y);
    S.O.P(z);
}
int c = a+b;
return c;
}
  
```

Class Launch

```

{ P.S.U.m (String arg[])
Calculator calc = new Calculator();
  
```

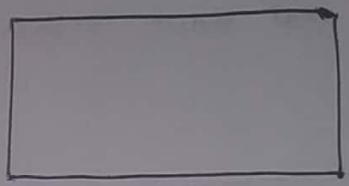
Activation record of disp()



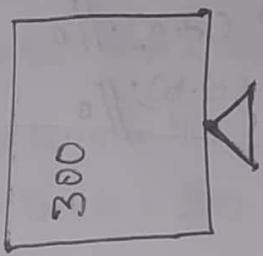
Activation record of disp()

Activation record of main()

Activation record of main()



Stack



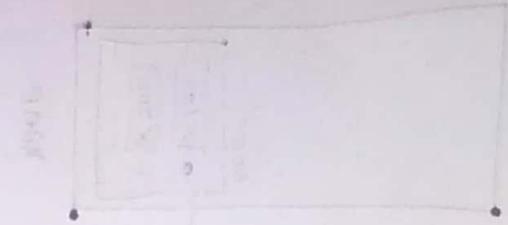
Stack

Code

[Instance variables Initialization by JVM]

* Class Demo

```
{  
    int a;  
    int b;  
}
```



Class Launch

```
{  
    public static void main(String args[]){  
        Demo d = new Demo();  
        System.out.println(d.a); // 0  
        System.out.println(d.b); // 0  
    }  
}
```

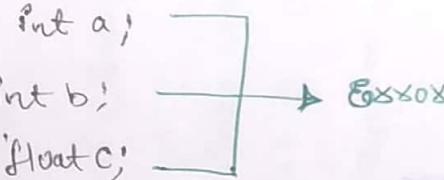


* [Local variables Initialization]

Class Launch

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

```
{  
    int a;  
    int b;  
    float c;  
}
```



```
        System.out.println(a);  
        System.out.println(b);  
        System.out.println(c);  
    }  
}
```

```
}  
}
```

Special characters allowed during variable initialization.

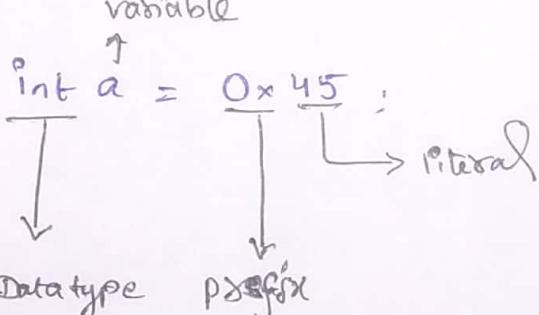
There are only two special characters that are allowed they are:- '-' and '\$'.

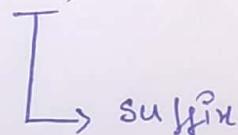
Legal

```
int temp = 40;  
int te-mp = 40;  
int t@ - - mp = 40;  
int --temp = 40;  
int temp - - = 40;  
int temp = 4_0;  
int temp = 4____0;
```

illegal

```
int #temp = 40;  
int temp = -40;  
int te@mp = 40;  
int temp = 40-;  
int temp = $40$40;  
int temp = 4$0;  
int temp = 40$;
```

* 
variable
↓
int a = 0x45 ;
↓ literal
↓
datatype prefix

* float b = 30.0f;

↓ suffix

Method Overloading :-

- In C language creating multiple functions with the same name is not possible, however any number of (methods) functions can be created but every function should have a different name.
- This is not programmer friendly because the programmer is now expected to remember multiple method names

```
* #include <stdio.h>

void main()
{
    int a = 85, b = 65, c = 45;
    float d = 35.5f, e = 45.5f; f = 60.6f;
    double g = 50.5; h = 60.5; i = 70.5;
}
```

~~Def void main~~

```
int add1( int x, int y )
{
    return x + y;
}
```

```
float add2( int x, int y )
{
    return x * y;
}
```

```
int add3 (int x, int y, int z)
{
    return x+y+z;
}
```

```
float add4 (float x, float y, float z)
{
    return x+y+z;
}
```

```
double add5 (double x, double y, double z)
{
    return x+y+z;
}
```

```
double add6 (int x, int y, double z)
{
    return x+y+z;
}
```

```
double add7 (double x, double y, int z)
{
    return x+y+z;
}
```

```
float add8 (float x, float y, int z)
{
    return x+y+z;
}
```

```
float add9 (int x, int y, float z)
{
    return x+y+z;
}
```

```
double add10 (double x, double y, float z)
{
    return x+y+z;
}
```

- In order to resolve the above shown problem in java we have a facility in creating multiple methods with the same name.
- The process of creating multiple methods with the same name inside a single class is called as method overloading
 (or) Compile time polymorphism. (or) ^{Ph} Virtual polymorphism
 (or) Early Binding.

* Class Launch

{

public static void main (String args [])

{

int a=35, b=65, c=45;

double e=35.6, h=60.5, i=70.5;

float d=35.5f, f=36.7f, g=80.6f;

Calculator calc = new Calculator();

S.o.p (calc.add(a,b,c));

S.o.p (calc.add(a,c));

}

}

Class Calculator

{

int add (int x, int y)

{ return x+y; }

float add (float x, float y)

{

 return x+y;

}

int add (int x, int y, int z)

{

 return x+y+z;

}

float add (float x, float y, float z)

{

 return x+y+z;

}

double add (double x, double y, double z)

{

 return x+y+z;

}

double add (int x, int y, double z)

{

 return x+y+z;

}

double add (double x, double y, int z)

{

 return x+y+z;

}

float add (float x, float y, float z)

{

 return x+y+z;

}

float add (int x, int y, float z)

{

 return x+y+z;

}

```

double add (double x, double y, float z)
{
    return x+y+z;
}

float add (float x, float y)
{
    return x+y;
}

```

Note :-

Method overloading helps us to achieve virtual polymorphism, Virtual Polymorphism is an illusion where the user feels that there is only and only one method performing multiple methods, activities which is not true in reality there are multiple methods each performing one particular activity.

Note: Nothing is overloaded in method overloading.

Numeric promotion in Method Overloading :-

→ Implicit type casting is also known as numeric promotions, where a smaller datatype is converted into a larger datatype and method overloading

Promotes numeric promotion.

* Class Calculator

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

```
{ return x+y;
```

```
}
```

```
double add (double x, double y)
```

```
{
```

```
return x+y;
```

```
}
```

```
}
```

Class Launch

```
{
```

```
public static void main (String args [])
```

```
{
```

```
Calculator calc = new Calculator();
```

```
calc.add (50.5f, 40.5f);
```

```
}
```

```
}
```

O/P :- 71.0

Note :-

===== return-type doesn't play a role in method overloading

and hence, the program shown below is results are

exxox.

Class calculator

```
{  
    int add (int x, int y) {  
        {  
            return x+y;  
        }  
    }
```

Error → void add (int x, int y)

```
{  
    int z = x+y;  
    S.O.P(z);  
}
```

Class Launch

```
{  
    PSVM Cstring CT)  
    {  
        calculator calc = new calculator();  
        calc.add(30, 40);  
    }  
}
```

Note: The program shown below is going to return an ambiguity error. because both the methods are capable of accepting integer values. Since method overloading promotes numeric promotion. Hence the effort put by the compiler to convert integers into float and double values is the case of both methods is same. Therefore it results in

Ambiguity.

* Class Calculator

{

double add(double x, float y)

{

return x+y;

}

double add (float x, double y)

{

return x+y;

}

Class Launch

{

param (String args [])

{

Calculator calc = new Calculator();

S. O. P (calc.add (20,30));

}

Ambiguity.

* Class Calculator

```
{  
    double add(double x, float y)  
    {  
        return x+y;  
    }  
  
    double add (float x, double y)  
    {  
        return x+y;  
    }  
}
```

Class Launch

```
{  
    public static void main (String args[])  
    {  
        Calculator calc = new Calculator();  
        System.out.println (calc.add (20, 30));  
    }  
}
```

~~1~~ Class Calculator

```
{  
    double add (double x, int y)  
    {  
        return x+y;  
    }  
}
```

```
float add (float x, float y)  
{  
    return x+y;  
}
```

class Launch

```
{  
}
```

```
PSvm (String [] args)
```

```
{
```

```
Calculator calc = new Calculator();
```

```
System.out.println (calc.add (30, 50)); // 80.0
```

```
}
```

~~X2~~ class Calculator

```
{
```

```
double add (float x, double y)
```

```
{
```

```
return x+y;
```

```
}
```

```
float add (float x, float y)
```

```
{
```

```
return x+y;
```

```
}
```

```
}
```

class Launch

```
{
```

```
PSVm (String args [])
```

```
{
```

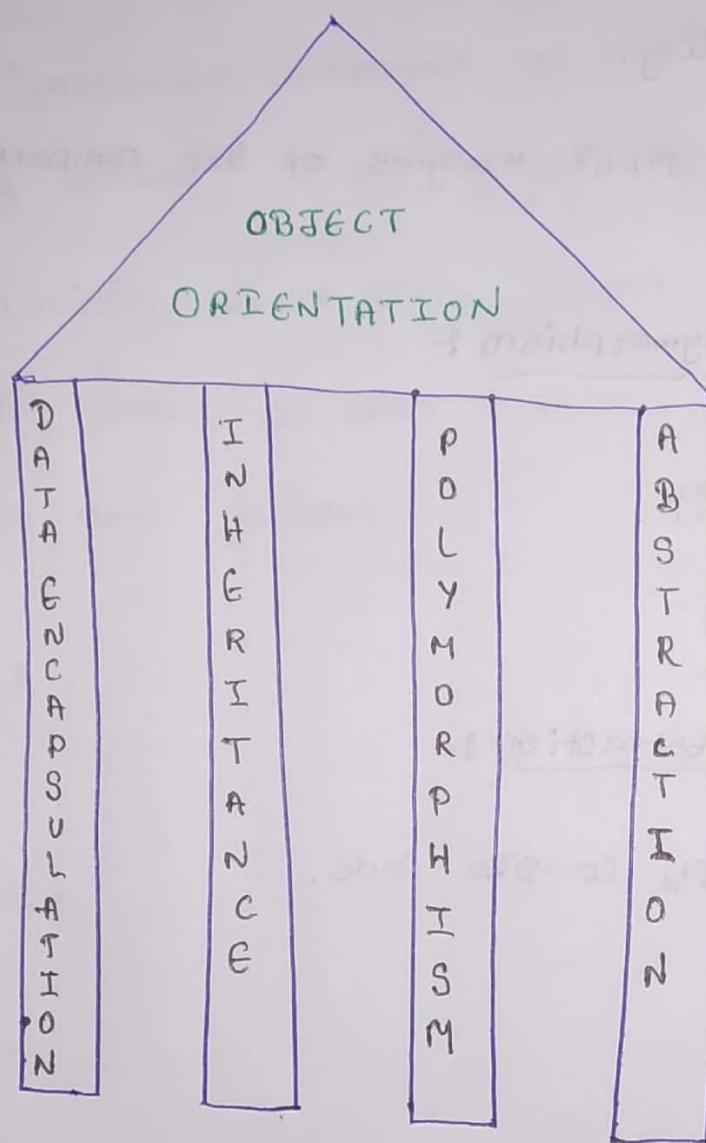
```
Calculator calc = new Calculator();
```

```
S.O.P (calc.add (30, 50)); // 80.0
```

```
}
```

```
}
```

Pillars of Object Orientation :-



→ Advantages of Data Encapsulation :-

1. Provides security and the access to the components of the class is controlled.
2. Easy maintainability.

Advantages of Inheritance :-

1. Reduces the efforts of the programmers.
2. Code reusability.
3. Increase the profit margins of the company.

Advantages of Polymorphism :-

1. Code reusability.
2. Code flexibility.
3. Code reduction.

Advantages of Abstraction :-

1. Helps us to write concise code.
2. Code reduction.

1. Data Encapsulation :-

Data Encapsulation is a process of providing security to the most important component of the class. i.e., data members by avoiding direct access using private access modifiers, avoiding direct access doesn't mean preventing direct access. Hence we also provide control access using setters and getters.

Direct Access :-

Class Demo

```
{  
    private int a;  
}
```

Class Launch

```
{  
    p s u m (String args[])  
    {  
        Demo d = new Demo();  
        d.a = 100;  
        s.o.p(d.a);  
    }  
}
```

Exception

Control access for

Class Demo

{

Private int a;

Public void setA (int x)

{

a = x;

}

Public int getA ()

{

return a;

}

}

Class Launch

{

psvm (String args [])

{

Dem0 d = new Dem0();

// d.a = 100;

d.setA (100);

// S.O.P(d.a);

S.O.P (d.getA());

}

}

O/P :- 100

* Class Demo

```
{  
    private int a;  
  
    public void setA(int x)  
    {  
        if (x >= 0)  
        {  
            a = x;  
        }  
        else  
        {  
            System.out.println("Please provide positive value");  
        }  
    }  
}
```

public int getA()

```
{  
    return a;  
}  
}
```

Class Launch

```
{  
    public static void main(String args[])  
    {  
        Demo d = new Demo();  
        d.setA(-100);  
        System.out.println(d.getA());  
    }  
}
```

O/P :- Please provide a positive value
0

Note:- A method can accept any number of parameters but however return only and only one value.

Class Dog

{

private String name;

private String breed;

private int cost;

public void set (String x, String y, int z)

{ name = x;

breed = y;

cost = z;

}

public String getName()

{

return name;

}

public String getBreed()

{

return breed;

}

public int cost()

{

return cost;

}

}

Class Launch

{

sum (String args[])

{

Dog d = new Dog();

d.set ("Rocky", "Husky", 9999)

S.O.P (d.getName());

S.O.P (d.getBreed());

S.O.P (d.getCost());

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

}

Shadowing Problem:

As shown in the below program the convention in the Java is the name of the local variables accepting the input should be same as the name of the instance variable but however this results in shadowing problem.

→ Shadowing problem is a phenomenon where the value collected inside the local variable is reassigned back to the same local variable due to the name conflict that occurs between the local and instance variables.

This name conflict occurring because the names of the instance and local variables are same.

Class Dog

{

private String name;
private String breed;
private int cost;

public void set (String name, String breed, int cost)

{

name = name;
breed = breed;
cost = cost;

}

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

```
public String getBreed()
{
    return breed;
}
```

```
public int getCost()
{
    return cost;
}
```

Class Launch

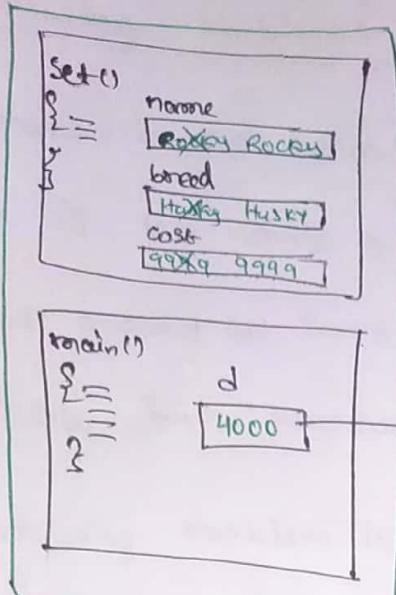
```
{ main()
{
    Dog d = new Dog();
    System.out.println(d.getName());
    System.out.println(d.getBreed());
    System.out.println(d.getCost());
}
```

```
Dog d = new Dog();
d.setName("Rocky");
d.setBreed("Husky");
d.setCost(9999);
System.out.println(d.getName());
System.out.println(d.getBreed());
System.out.println(d.getCost());
```

```
}
```

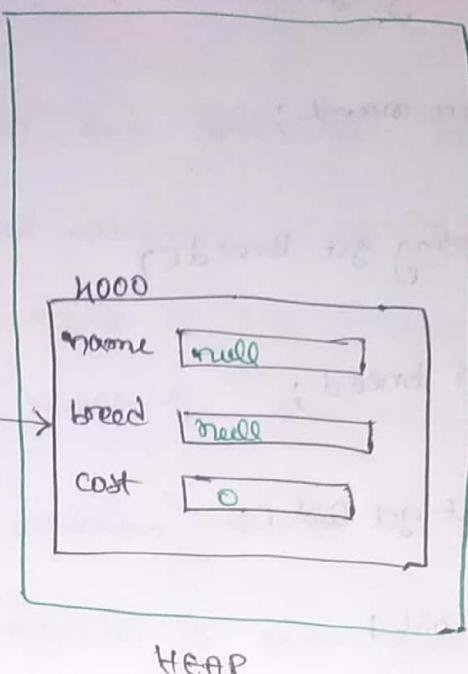
O/P :-
null
null
0

AR
of
Set()



AR of
main()

STACK



HEAP

→ As shown in the below program in order to resolve the shadowing problem we make use of 'this' keyword.

If we use this keyword the associated variable is treated as instance variable. Hence there is no name conflict and in this case data present in local variable will be assigned to instance variable.

* Program.

```
Class Dog  
{  
private String name;  
private String breed;  
private int cost;
```

}

```
public void set (String name, String breed, int cost)
```

```
{
```

```
    this.name = name;
```

```
    this.breed = breed;
```

```
    this.cost = cost;
```

```
}
```

```
public String getName ()
```

```
{
```

```
    return Name;
```

```
}
```

```
public String get Breed ()
```

```
{
```

```
    return breed;
```

```
}
```

```
public int get Cost ()
```

```
{
```

```
    return cost;
```

```
}
```

```
}
```

```
class Launch
```

```
{
```

```
    PSum (String args [])
```

```
{
```

```
    Dog d = new Dog ();
```

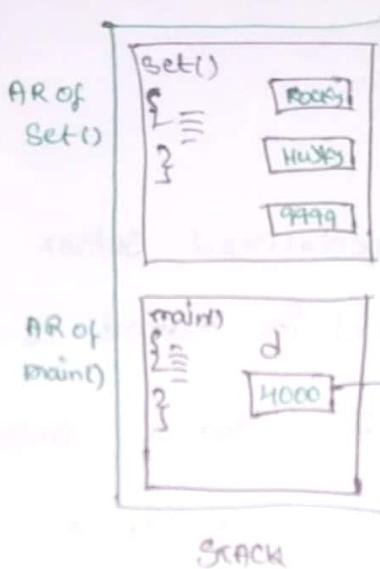
```
    d.set ("Rocky", "Husky", 9999);
```

```
    S.O.P (d.getName());
```

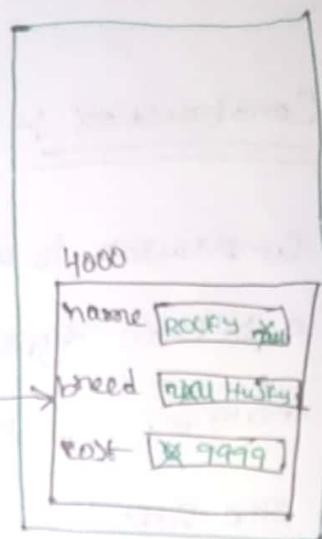
```
    S.O.P (d.getBreed());
```

```
    S.O.P (d.getCost());
```

```
}
```



Stack



Heap

O/P

Rocky

Husky

9999

* this. Key word is used to resolve shadowing problem and it is also used to point towards currently executing object.

* Constructors :-

- Constructor is a specialized setter which doesn't have a return type. and its name is same as the class name. It is called during object creation.
- The first line of every constructor by default is a "Super()" call method only if "this()" call is not present.

* this. Key word is used to resolve shadowing problem and it is also used to point towards currently executing object.

* Constructors :-

- Constructor is a specialized setter which doesn't have a return type. and its name is same as the class name. It is called during object creation.
- The first line of every constructor by default is a "Super()" call method only if "this()" call is not present.

Note:- If the programmer doesn't include even a single constructor then JVM includes one 'zero' parameterised constructor called default constructor.

JVM is going to include a zero parameterised default constructor "only if only if the programmer doesn't include even a single constructor." even if the programmer includes one constructor then a default constructor will not be assigned (or) created.
Hence we have come across an error as below shown program - 11.

Class Dog extends Object

 Public int getCost()

 {
 return cost;
 }

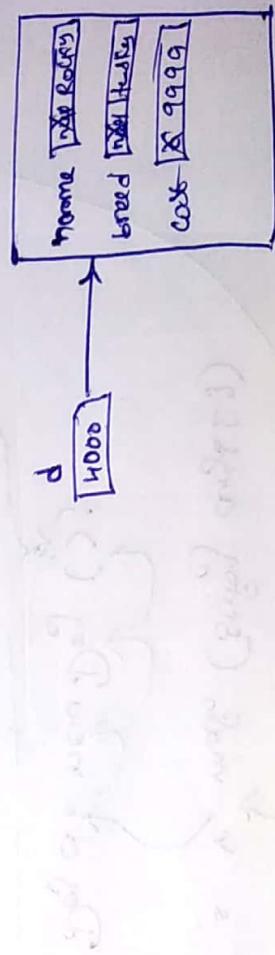
 private String name;
 private String breed;
 private int cost;

 public Dog (String name, String breed, int cost)
 {
 super();
 this.name = name;
 this.breed = breed;
 this.cost = cost;
 }

 Public String getName()

 {
 return name;
 }

 Public String getBreed()
 {
 return breed;
 }



Class Dog extends Object

```
{  
    private String name;  
    private String breed;  
    private int cost;
```

public Dog()

super()

D. object()

==
==

class launch

R = new Dog args()

{
 R = new Dog();

S.O.P(R.get name());

S.O.P(R.get breed());

S.O.P(R.get cost());

? ? ?

public String get Name()

```
{  
    return no name;
```

```
{  
    public String get breed()  
    {  
        return breed;
```

O/P == null
null

o

class launch

public int get cost()

class launch

3

Program-11

class Dog

{

 private String name;

 private String breed;

 private int cost;

 public Dog (String name, String breed)

{

 super();

 this.name = name;

 this.breed = breed;

}

 public String getName()

{

 return name;

}

 public String getBreed()

{

 return breed;

}

 public int getCost()

{

 return cost;

}

}

class Launch

{

 public void main (String [] args)

{

 Dog d1 = new Dog(); → Error

 System.out.println(d1.getName());

 System.out.println(d1.getBreed());

 System.out.println(d1.getCost());

Dog d2 = new Dog("Lobo", "Pug");

System.out.println(d2.getName());

System.out.println(d2.getBreed());

→ Constructor Chaining is a process where a child class constructor calls the Parent class constructor through the 'super()' method.

Constructor Overloading :-

- * It is a process of having multiple constructors inside a single class. This, however, results in a confusion. In order to resolve the confusion, the same cases taken into consideration in method overloading are considered here as well. They are
 - 1) No. of parameters.
 - 2) Datatype of parameters.
 - 3) Order of data types.

Class Dog Extends Object
{

 private String name;

 private String breed;

 private int cost;

 public Dog()

 { super();
 name = "Rocky";

 breed = "Pug";

 cost = 9999;

}

```
public String getName()
```

```
{
```

```
    return name;
```

```
}
```

```
public String get Breed()
```

```
{
```

```
    return Breed;
```

```
}
```

```
public int get cost()
```

```
{
```

```
    return cost;
```

```
}
```

```
public Dog (String name, String breed, int cost)
```

```
{
```

```
    super();
```

```
    this.name = name;
```

```
    this.breed = breed;
```

```
    this.cost = cost;
```

```
}
```

```
{
```

```
class Launch extends object
```

```
{
```

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

```
    S.O.P (d1.getName()); // ROCKY
```

```
    S.O.P (d1.get Breed()); // PUG
```

```
    S.O.P (d1.get Cost()); // 9999
```

```
    Dog d2 = new Dog ("Lobo", "Husky", 8888);
```

```
    S.O.P (d2.getName()); // Lobo
```

```
    S.O.P (d2.get Breed()); // Husky
```

```
    S.O.P (d2.get Cost()); // 8888
```

```
}
```

```
{
```

Local Chaining:

Local chaining is a process of one constructor calling another constructor present in ^{same} class using 'this()' call. If 'this()' method is present in a constructor then Super() will not be present.

```

class Dog {
    private String name;
    private String breed;
    private int cost;

    public Dog() {
        Super();
        name = "Rocky";
        breed = "Pug";
        cost = 9999;
    }

    public String getName() {
        return name;
    }

    public String getBreed() {
        return breed;
    }

    public int getCost() {
        return cost;
    }

    public Dog (String name, String breed, int cost) {
        this();
        this.name = name;
        this.breed = breed;
        this.cost = cost;
    }
}

```

(05)

```

class Launch {
    PSVM CString args[])

    {
        Dog d1 = new Dog ("Tommy",
                           "Pitbull",
                           7777);

        System.out.println(d1.getName());
        System.out.println(d1.getBreed());
        System.out.println(d1.getCost());
    }
}

```

name	Tommy	Pitbull
breed	Pitbull	Pitbull
cost	7777	7777

Local Chaining : 89-2

Class Dog

```
{ private string name;  
private string breed;  
private int cost;
```

public Dog()

```
{ this.name = "Rocky";  
this.breed = "Pug";  
this.cost = 999; }
```

public Dog(string name, string breed, int cost)

```
{ super();
```

```
this.name = name;  
this.breed = breed;  
this.cost = cost; }
```

public Dog(string name, string breed, int cost)

```
{ this.c =;
```

```
this.name = cost;  
this.breed = cost; }
```

public string getName()

```
{ return name; }
```

public string getBreed()

```
{ return breed; }
```

public int getCost()

```
{ return cost; }
```

public void setBreed(string breed)

```
{ breed =;
```

```
this.breed = breed; }
```

public void setCost(int cost)

```
{ cost =;
```

```
this.cost = cost; }
```

Output

Rocky

Pug

999

Dog d = new Dog("Tommy", "Husky", 999);

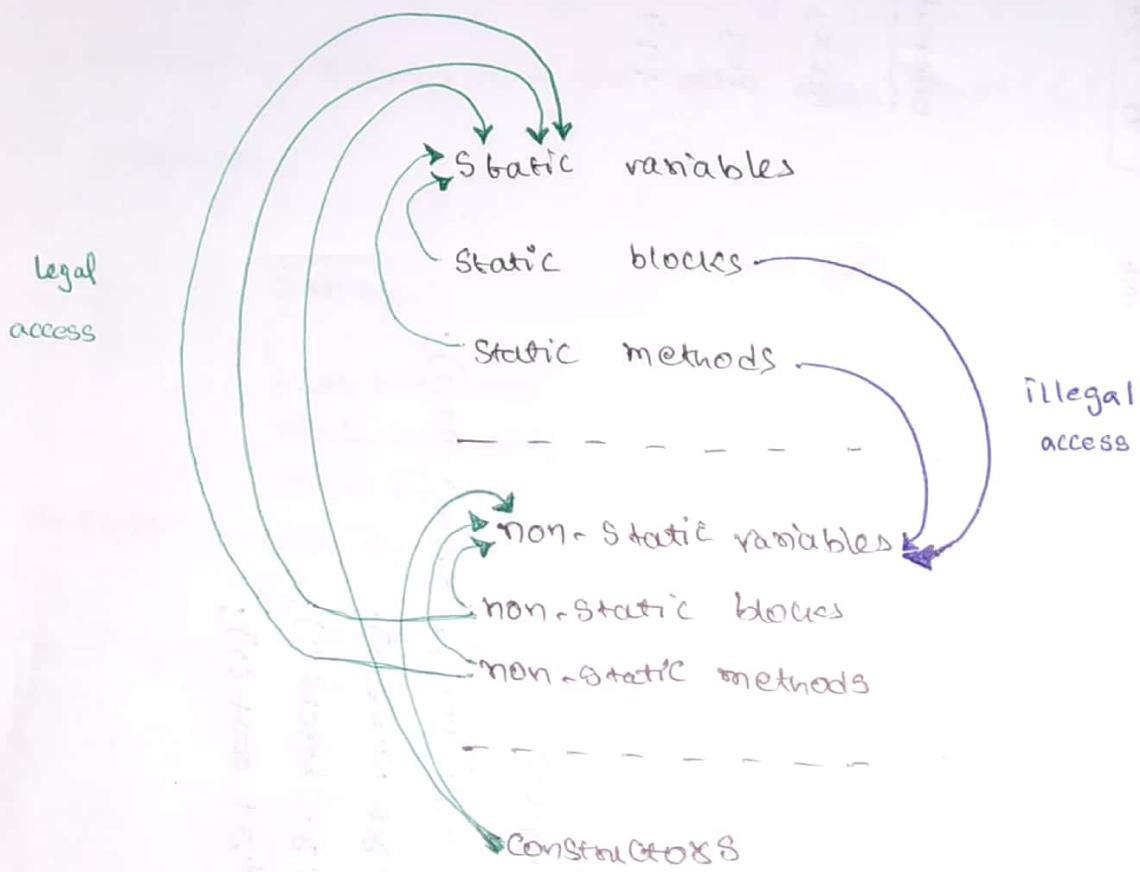
```
5. o.p(d.get Name());
```

```
5. o.p(d.get Breed());
```

```
5. o.p(d.get Cost());
```

5

STATIC :-



- JVM always executes static variables first, then static blocks after that it executes main method. Inside the main method if a non-static method is called first it gets executed or if a static method is called first it gets executed hence, the order of execution of static & non-static methods totally depends on how the methods are being called.

Irrespective of the number of methods present inside a program main method is always called first.

CODE

```
class Demo
{
    static int a,b,c;
    static {
        a=10;
        b=20;
        c=30;
    }
    public static void disp1()
    {
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
    int xyz;
    {
        System.out.println("Inside a non static block");
    }
    public void disp2()
    {
        System.out.println(x);
        System.out.println(y);
        System.out.println(z);
    }
}
```

public Demo()

{

x=100;

y=200;

z=300;

} }

class Launch

{

public static void main

(String args[])

{

Demo d=new Demo();

d.disp2();

} }

O/P :-

10

20

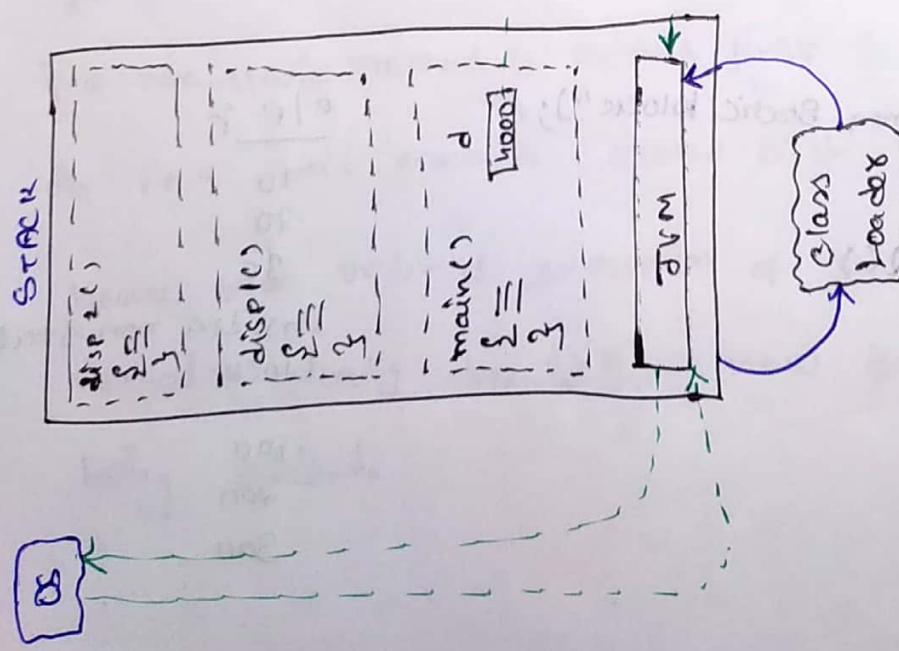
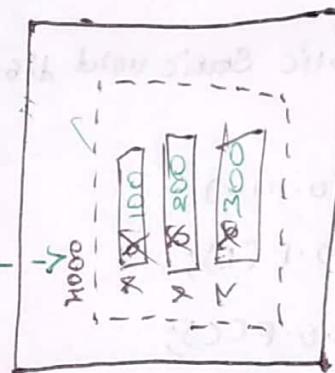
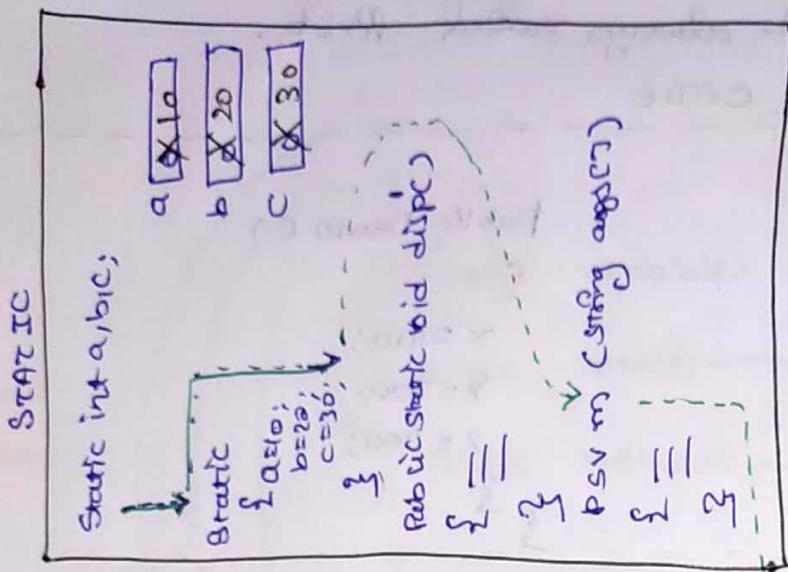
30

Inside a non static block

100

200

300



static variables



static blocks



main method

Advantages of static block :-

class Launch

{

Static

{

System.out.println("ABC");

}

System.out.println("Hello");

{

System.out.println("World");

}

}

O/P :- ABC

Hello

→ Static blocks can be used to initialize static variables.

The advantage of a static block is its execution happens

even before main method. Hence if any part of the

code has to be executed even before the main method

executes it should be included inside a static block.

Advantage of Static variables :-

class Examex

{
int amount;

float t;

float si;

float x;

public void acceptInput ()

{
System.out.println("Enter the required amount")

Scanner scan = new Scanner (System.In)

amount = scan.nextInt();

System.out.println("Enter the time Required");

t = scan.nextFloat();

x = 2.5f;

}

public void compute ()

{
SI = amount * t * x / 100;
}

public void disp()

{
System.out.println("The simple interest is: " + SI);
}

}

Class Launch

```
{ psum(String args[])
```

```
{ Farmer f1 = new Farmer();
```

```
Farmer f2 = new Farmer();
```

```
Farmer f3 = new Farmer();
```

```
f1.acceptInput();
```

```
f1.compute();
```

```
f1.display();
```

```
f2.acceptInput();
```

```
f2.compute();
```

```
f2.display();
```

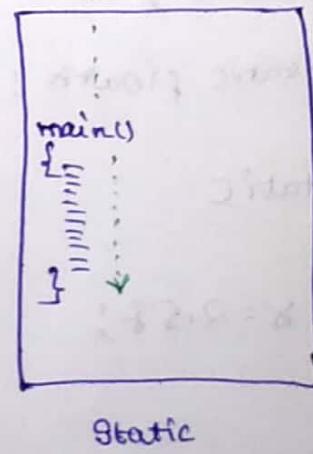
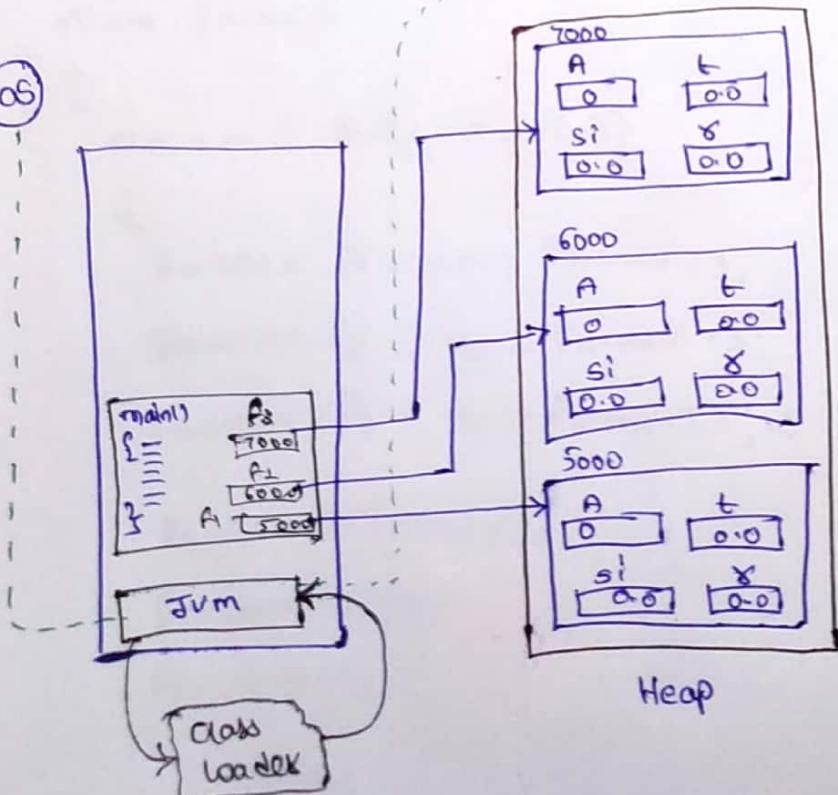
```
f3.acceptInput();
```

```
f3.compute();
```

```
f3.display();
```

```
}
```

(05)



As shown in the above program the rate of interest for all the farmer objects is common, since it is a non-static instance variable it is allocated memory every time an object gets created. Since the data is common allocating memory multiple times again and again for every object will result in inefficient memory management. Hence in order to create and maintain a common copy of rate of interest we will declare the variable as static. By declaring the variable as static memory is allocated only once to the static segment this results in efficient memory utilization as shown in the below program.

Class Farmer

```
{  
    float amount;  
    float t;  
    float si;  
    static float r;  
    static  
{  
    r=2.5f;  
}
```

```
public void accept Input()
```

```
{
```

```
    S.O.P ("Enter the required amount")
```

```
Scanner scan = new Scanner (System.In);
```

```
amount = scan.nextInt();
```

```
S.O.P ("Enter the time required");
```

```
t = scan.nextFloat();
```

```
}
```

```
public void compute()
```

```
{
```

```
SP = (amount * t * x) / 100;
```

```
}
```

```
public void disp()
```

```
{
```

```
S.O.P ("The simple interest is : " + " + 5% );
```

```
}
```

```
}
```

```
Class Launch
```

```
{
```

```
PSVM (String args[])
```

```
{
```

```
farmer f1 = new Farmer();
```

```
farmer f2 = new Farmer();
```

```
farmer f3 = new Farmer();
```

```
f1.acceptInput();
```

```
f1.compute();
```

```
f1.disp();
```

f2.acceptInput();

f2.compute();

f2.display();

f3.acceptInput();

f3.compute();

f3.display();

}

STACK

main()	R3 70000
	R2 60000
	R1 50000

JVM

class loader

2000	amount	b
	615000	4
	1000 1500.0	

6000	amount	b
	675000	1
	18 750	

5000	amount	b
	650,000	2
	2500.0	

static Root S;	↓
Static	↓
↓	5=2.5f;
↓	3=
maing()	↓
↓	3=

STATIC

class A {

 int a = 10;

 int b = 20;

 int c = 30;

 int d = 40;

 int e = 50;

 int f = 60;

 int g = 70;

 int h = 80;

 int i = 90;

 int j = 100;

 int k = 110;

 int l = 120;

 int m = 130;

 int n = 140;

 int o = 150;

 int p = 160;

 int q = 170;

 int r = 180;

 int s = 190;

 int t = 200;

 int u = 210;

 int v = 220;

 int w = 230;

 int x = 240;

 int y = 250;

 int z = 260;

 int aa = 270;

 int bb = 280;

 int cc = 290;

 int dd = 300;

 int ee = 310;

 int ff = 320;

 int gg = 330;

 int hh = 340;

 int ii = 350;

 int jj = 360;

 int kk = 370;

 int ll = 380;

 int mm = 390;

 int nn = 400;

 int oo = 410;

 int pp = 420;

 int qq = 430;

 int rr = 440;

 int ss = 450;

 int tt = 460;

 int uu = 470;

 int vv = 480;

 int ww = 490;

 int xx = 500;

 int yy = 510;

 int zz = 520;

 int aa = 530;

 int bb = 540;

 int cc = 550;

 int dd = 560;

 int ee = 570;

 int ff = 580;

 int gg = 590;

 int hh = 600;

 int ii = 610;

 int jj = 620;

 int kk = 630;

 int ll = 640;

 int mm = 650;

 int nn = 660;

 int oo = 670;

 int pp = 680;

 int qq = 690;

 int rr = 700;

 int ss = 710;

 int tt = 720;

 int uu = 730;

 int vv = 740;

 int ww = 750;

 int xx = 760;

 int yy = 770;

 int zz = 780;

 int aa = 790;

 int bb = 800;

 int cc = 810;

 int dd = 820;

 int ee = 830;

 int ff = 840;

 int gg = 850;

 int hh = 860;

 int ii = 870;

 int jj = 880;

 int kk = 890;

 int ll = 900;

 int mm = 910;

 int nn = 920;

 int oo = 930;

 int pp = 940;

 int qq = 950;

 int rr = 960;

 int ss = 970;

 int tt = 980;

 int uu = 990;

 int vv = 1000;

 int ww = 1010;

 int xx = 1020;

 int yy = 1030;

 int zz = 1040;

 int aa = 1050;

 int bb = 1060;

 int cc = 1070;

 int dd = 1080;

 int ee = 1090;

 int ff = 1100;

 int gg = 1110;

 int hh = 1120;

 int ii = 1130;

 int jj = 1140;

 int kk = 1150;

 int ll = 1160;

 int mm = 1170;

 int nn = 1180;

 int oo = 1190;

 int pp = 1200;

 int qq = 1210;

 int rr = 1220;

 int ss = 1230;

 int tt = 1240;

 int uu = 1250;

 int vv = 1260;

 int ww = 1270;

 int xx = 1280;

 int yy = 1290;

 int zz = 1300;

 int aa = 1310;

 int bb = 1320;

 int cc = 1330;

 int dd = 1340;

 int ee = 1350;

 int ff = 1360;

 int gg = 1370;

 int hh = 1380;

 int ii = 1390;

 int jj = 1400;

 int kk = 1410;

 int ll = 1420;

 int mm = 1430;

 int nn = 1440;

 int oo = 1450;

 int pp = 1460;

 int qq = 1470;

 int rr = 1480;

 int ss = 1490;

 int tt = 1500;

 int uu = 1510;

 int vv = 1520;

 int ww = 1530;

 int xx = 1540;

 int yy = 1550;

 int zz = 1560;

 int aa = 1570;

 int bb = 1580;

 int cc = 1590;

 int dd = 1600;

 int ee = 1610;

 int ff = 1620;

 int gg = 1630;

 int hh = 1640;

 int ii = 1650;

 int jj = 1660;

 int kk = 1670;

 int ll = 1680;

 int mm = 1690;

 int nn = 1700;

 int oo = 1710;

 int pp = 1720;

 int qq = 1730;

 int rr = 1740;

 int ss = 1750;

 int tt = 1760;

 int uu = 1770;

 int vv = 1780;

 int ww = 1790;

 int xx = 1800;

 int yy = 1810;

 int zz = 1820;

 int aa = 1830;

 int bb = 1840;

 int cc = 1850;

 int dd = 1860;

 int ee = 1870;

 int ff = 1880;

 int gg = 1890;

 int hh = 1900;

 int ii = 1910;

 int jj = 1920;

 int kk = 1930;

 int ll = 1940;

 int mm = 1950;

 int nn = 1960;

 int oo = 1970;

 int pp = 1980;

 int qq = 1990;

 int rr = 2000;

 int ss = 2010;

 int tt = 2020;

 int uu = 2030;

 int vv = 2040;

 int ww = 2050;

 int xx = 2060;

 int yy = 2050;

 int zz = 2060;

 int aa = 2070;

 int bb = 2080;

 int cc = 2090;

 int dd = 2100;

Legal + Illegal Naming conventions of variables

In the case of variable name two special characters are allowed '`$`' and '`_`'. They can be used any number of times during the variable creation. They can be used before, after and in between variable names.

But, in the case of literal only and only one special character is permitted which is underscore '`_`'. It can be used only in between the literal using underscore '`_`' before or after the literal is not permitted. However it can be used any number of times in between the literals.

Legal

```
int temp = 0-45;
```

```
int temp = 0b11_11;
```

```
float temp = 45.5F;
```

illegal

```
int temp = 0b-1111;
```

```
int temp = -0b1111;
```

```
int temp = 0-b1111;
```

```
long temp = 45-2;
```

```
long temp = 45L-;
```

illegal

```
int temp = -045;
```

```
int temp = _0x45;
```

```
int temp = 0-x45;
```

```
int temp = 0x-45;
```

```
int temp = 0b45;
```

```
float temp = 45.5_f;
```

```
float temp = 45.5f-;
```

```
float temp = 45.-5f;
```

```
float temp = 45_.5f;
```

import java.util.StringTokenizer;

Class Launch

```
{ public sum (String args[])
```

```
{
```

```
String a = "ABC FOR TECHNOLOGY TRAINING";
```

```
String StringTokenizer st = new StringTokenizer(a, " ");
```

```
while (st.hasMoreTokens()) = true
```

```
.
```

```
s.o.p (st.nextToken());
```

```
{ }  
{ }
```

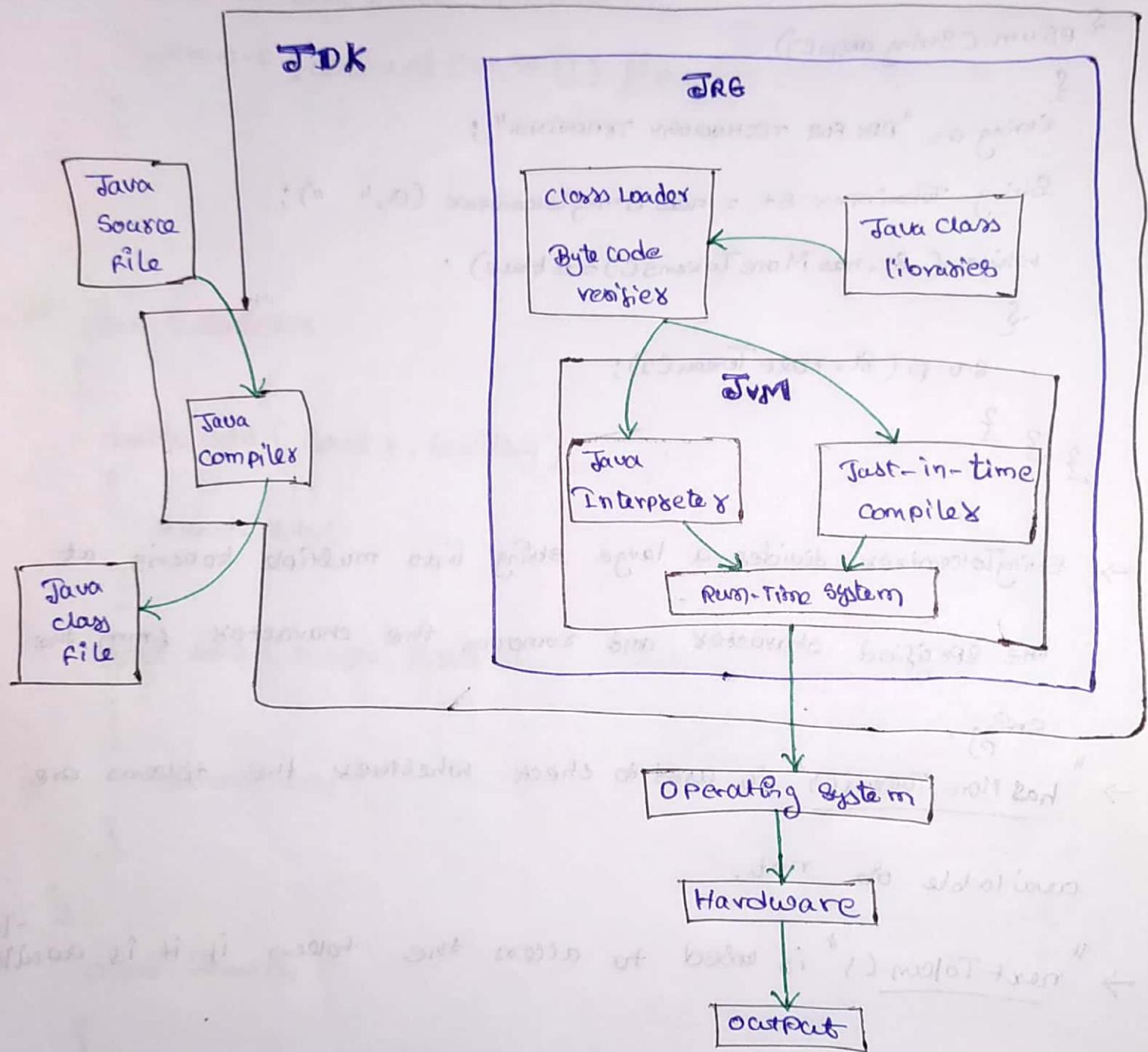
→ StringTokenizer divides a large string into multiple tokens at the specified character and removes the character from the

String.

→ "hasMoreTokens()" is used to check whether the tokens are available or not.

→ "nextToken()" is used to access the token if it is available.

Architecture of Java



Static variables :-

Static variables are also called as class variables because memory for static variables is allocated once to the static segment and the same copy of static variable can be used by all the methods and the objects of the class.

Applications of Static :-

Class Car

{
 Private int mileage;

 Public static void disp()

 {
 S.o.p (" your car is beautiful ");

}

 Public void mileage ()

{

 Scanner Scan = new Scanner (System.in);

 S.o.p (" Enter Mileage ");

 mileage = Scan.nextInt();

 if (mileage > 15)

{

 S.o.p (" The mileage is great ");

{

 else

{

 S.o.p (" The mileage is less ");

Application of Static variable:

In order to describe and store such properties which are common to all the objects of class we use static variables.

Advantage: Better & efficient memory utilization.

Application of Static methods:-

→ In order to describe and invoke such behaviours which are common to all the objects of a class we make use of static methods.

→ Static Methods always access only the common properties of a class. In other words, "STATIC methods always access static properties". If the output of the method is same to all the objects of the class then such methods should be declared as static.

Advantage: Output of a static method is always same irrespective of the object.

Application of static blocks:-

If some part of the code has to be executed even before the main method is executed then

that part of the code should be included ~~in~~

static block and both static block and main method

have to be present in the same class.

Advantage: Execution of static block happens even before the execution of the main method.

Inheritance

- Inheritance is a process of coding a java project or program in the form of hierarchy of classes.
It is a process where the data and properties present in one class is inherited to another class.
- A class from where the data was inherited is known as parent class or super class. The class into which the data and properties got inherited is called as the child class or the sub class.

Company - 1

6 months

1 cx

cricketers

name
age
ph.no
height
weight
Centuries
Duck outs
avg
Wkts
not outs
Strike rate

Footballers

name
age
ph.no
height
weight
goals
Penalties
red card
Yellow card
corner kicks

6 months

1 cx

Company - 2

6 months

1 cx

players

name
age
ph.no
height
height

3 months

2.5 cx

cricketers

Centuries
avg
Wkts
Duck outs
Strike rate

Footballers

goals
corner kicks
red cards
Penalties
Yellow cards

Rules of Inheritance :-

①

Private data members of a class (or) private properties of a class can't be inherited. Constructors also can't be inherited.

```

Class You
{
    Private int acc-no;
    Private int Pwd;

    Public void y()
    {
        acc-no = 2222;
        Pwd = 7777;
    }
}
  
```

```

Class Friend extends You
{
    Public void change Data()
    {
        acc-no = 3333;
        Pwd = 8888;
    }

    Public void disp()
    {
        S.O.P(acc-no);
        S.O.P(Pwd);
    }
}
  
```

Exxox

```

Class Launch
{
    PsVm(String ...[])
    {
        Friend f = new Friend();
        f.changeData();
        f.disp();
    }
}
  
```

Rule-2 :-

Constructors can't be inherited however they can be executed using super()

Class You extends Object

{

 private int acc-no;

 private int pwd;

 class Object

 public You();

 class Object

 super();

 S.o.P ("Parent constructor is
executed by super()");

}

 class Friend extends You

{

 public Friend()

{

 super();

}

 class Launch

{

 psvm(String args[])

{

 Friend f = new Friend();

}

Output

Parent constructor is
executed by super()

Rule-3 :-

Multiple Inheritance is not permitted in Java because it results in ambiguity and we come across diamond shape problem.

Ex:-

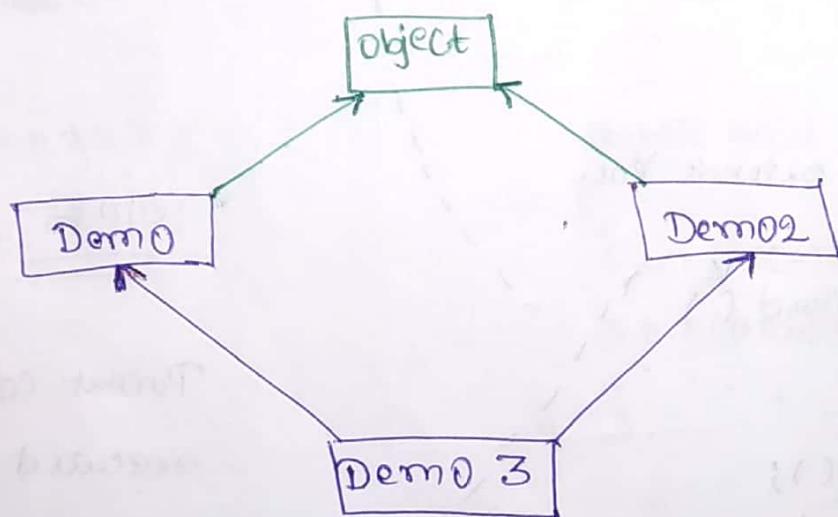
```
class Demo { int i=9; }
```

```
class Demo2 { int i=99; }
```

O/P: Error

```
Demo, Demo2 extends class Launch
{
    public void disp()
    {
        System.out.println(i);
    }
}
```

```
class Demo3 extends Demo, Demo2
{
    public void disp()
    {
        Demo d3 = new Demo();
        d3.disp();
    }
}
```



Rule - 4:

Multilevel inheritance is allowed in Java.

```

class Demo {
    class Demo2 {
        public void fun() {
            System.out.println("Hello");
        }
    }
    public void fun() {
        System.out.println("ABC");
    }
}

```

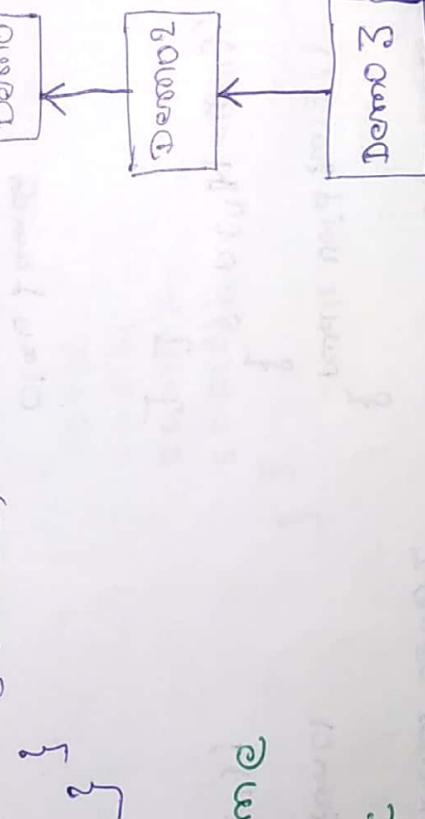
class Demo3

```
{ public void fun() {
    System.out.println("C");
}}
```

```

{
    Demo3 d3 = new Demo3();
    d3.fun();
}

```

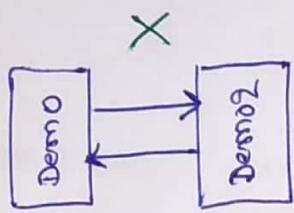


Op:

A B C

Rule - 5

Cyclic Inheritance is not permitted in Java.



class Demo extends Demo

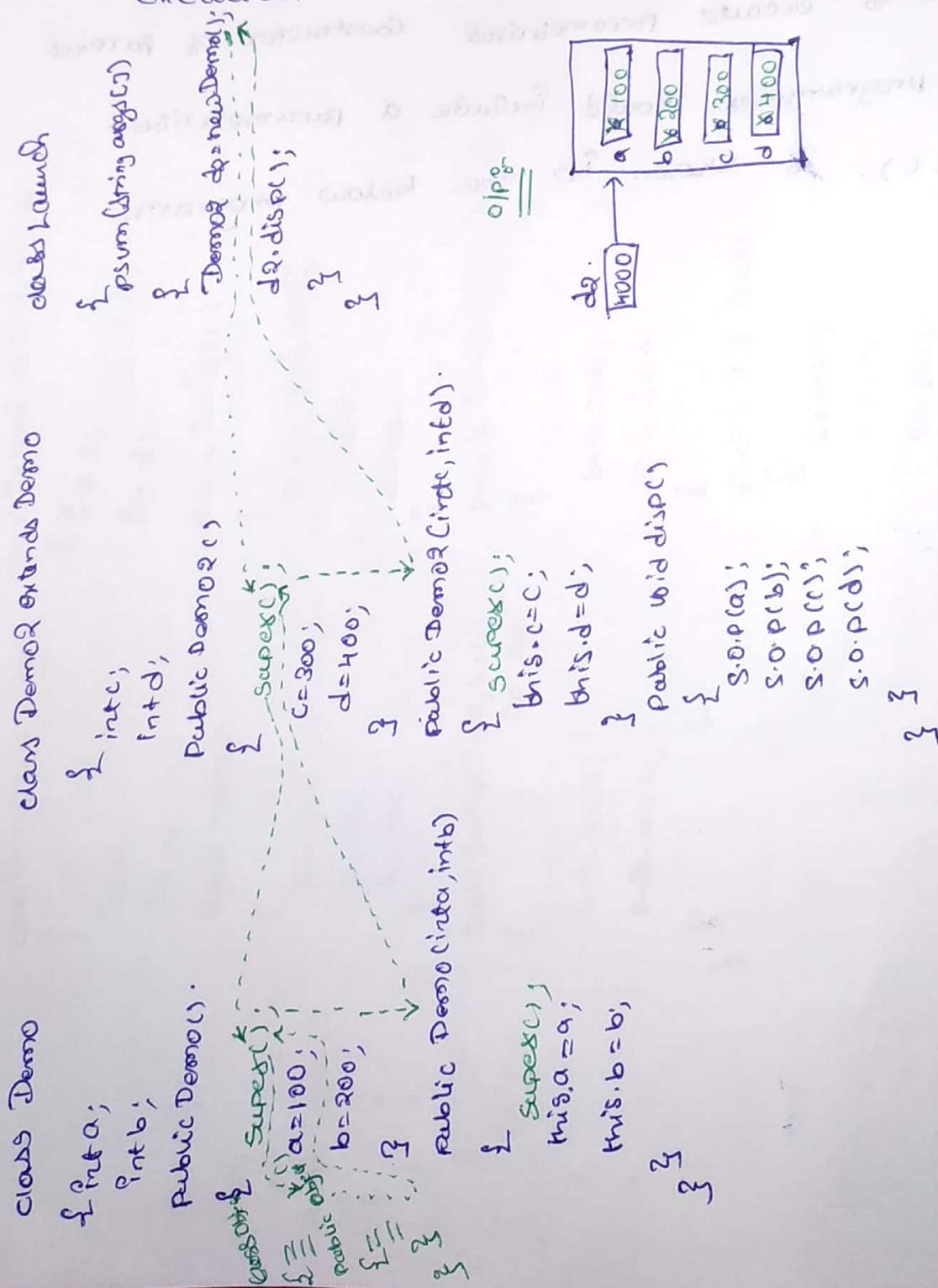
```
{ public void fun()
{ System.out.println("ABC");
}
}
```

Class Demo2

```
{ public void fun()
{ Demo d = new Demo();
d.fun();
}
}
```

Execution of Constructors with respect to Inheritance :-

Note :- The execution of the child class constructor occurs only once the execution of a parent class constructor is completed. In other words Parent class constructor always executes first later the child class constructor executes.



Notes

2. Irrespective of whether the child class constructor is parameterised (b) zero parameterised. If the Super() call present in it is zero parameterised always zero parameterised Parent class constructor will be executed.
3. In order to execute parameterised constructor of parent class programmers should include a parameterised Super() as shown in the below program.

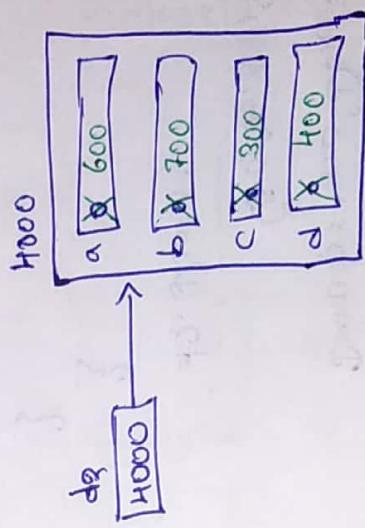
```
class Demo extends Object
```

```
class Demo2 extends Demo
```

```
{ int a;  
    int b;  
    public Demo()  
    {  
        Super();  
        a=100;  
        b=200;  
    }  
  
    public Demo(int a, int b)  
    {  
        Super();  
        this.a=a;  
        this.b=b;  
    }  
  
    public void disp()  
    {  
        System.out.println("a = "+a);  
        System.out.println("b = "+b);  
    }  
}
```

```
class Object  
{  
    public Object()  
    {  
        Super();  
        c=300;  
        d=400;  
    }  
  
    public Object2(int c, int d)  
    {  
        Super();  
        this.c=c;  
        this.d=d;  
    }  
  
    public void disp()  
    {  
        System.out.println("c = "+c);  
        System.out.println("d = "+d);  
    }  
}
```

```
class Demo2 extends Demo  
{  
    int c;  
    int d;  
    public Demo2()  
    {  
        Super(600, 700);  
        c=300;  
        d=400;  
    }  
  
    public void disp()  
    {  
        System.out.println("a = "+a);  
        System.out.println("b = "+b);  
        System.out.println("c = "+c);  
        System.out.println("d = "+d);  
    }  
}
```



```
class Demo2  
{  
    int a, b, c, d;  
    public Demo2()  
    {  
        Super();  
        a=600;  
        b=700;  
        c=300;  
        d=400;  
    }  
  
    public void disp()  
    {  
        System.out.println("a = "+a);  
        System.out.println("b = "+b);  
        System.out.println("c = "+c);  
        System.out.println("d = "+d);  
    }  
}
```

Class Demo

Public void disp()

```
{  
    int a;  
    int b;  
    int c;  
    int d;  
  
    public Demo()  
    {  
        this(99, 88);  
        a = 100;  
        b = 200;  
    }  
  
    public Demo(int a, int b)  
    {  
        super();  
        this.a = a;  
        this.b = b;  
    }  
  
    public Demo(int a, int b, int c, int d)  
    {  
        super();  
        this.a = a;  
        this.b = b;  
        this.c = c;  
        this.d = d;  
    }  
}
```

Class Demo2 extends Demo

```
{  
    public Demo2()  
    {  
        super(a);  
        super(b);  
        super(c);  
        super(d);  
        d = 400;  
    }  
  
    public Demo2(int a, int b)  
    {  
        super();  
        this.a = a;  
        this.b = b;  
    }  
  
    public Demo2(int a, int b, int c)  
    {  
        super();  
        this.a = a;  
        this.b = b;  
        this.c = c;  
    }  
}
```

Class Demo2

```
{  
    public Demo2()  
    {  
        super();  
        this.a = 100;  
        this.b = 200;  
        this.c = 300;  
        this.d = 400;  
    }  
  
    public Demo2(int a, int b)  
    {  
        super();  
        this.a = a;  
        this.b = b;  
    }  
  
    public Demo2(int a, int b, int c)  
    {  
        super();  
        this.a = a;  
        this.b = b;  
        this.c = c;  
    }  
}
```

Class Demo2

```
{  
    public Demo2()  
    {  
        super();  
        this.a = 100;  
        this.b = 200;  
        this.c = 300;  
        this.d = 400;  
    }  
  
    public Demo2(int a, int b)  
    {  
        super();  
        this.a = a;  
        this.b = b;  
    }  
  
    public Demo2(int a, int b, int c)  
    {  
        super();  
        this.a = a;  
        this.b = b;  
        this.c = c;  
    }  
}
```

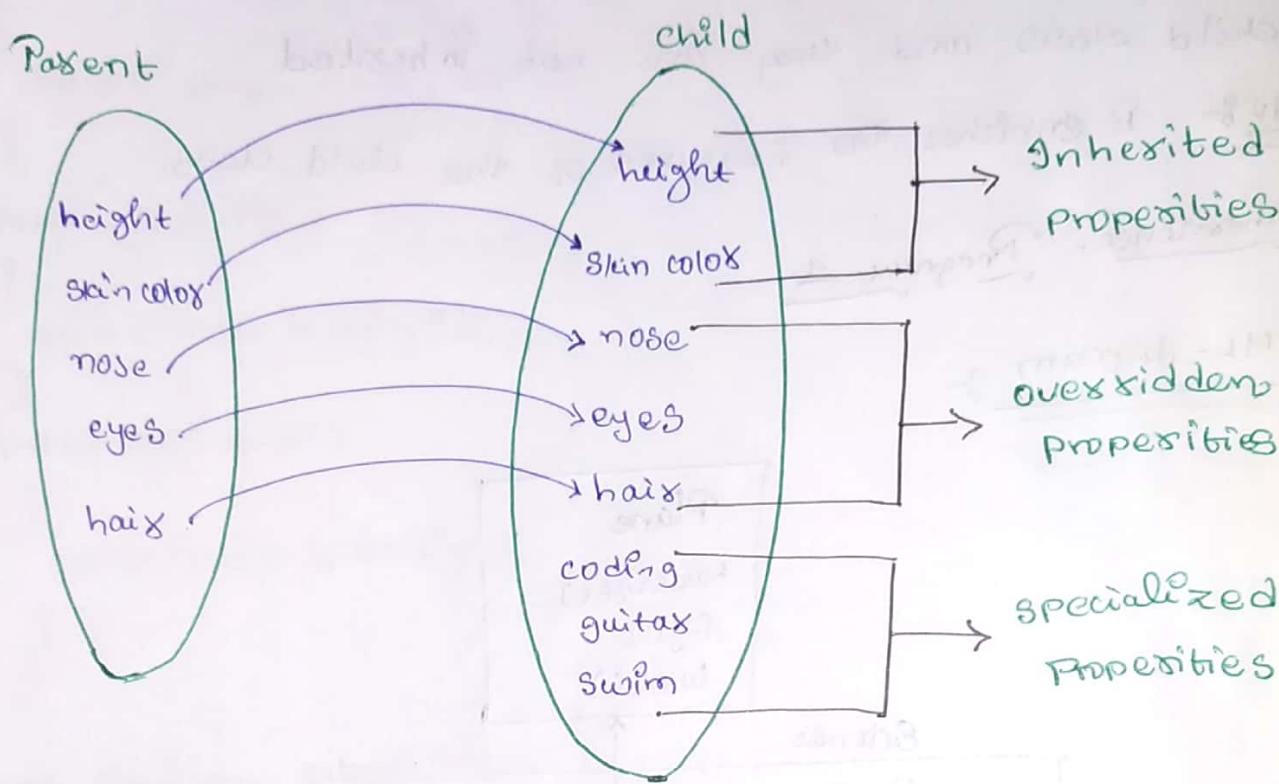
Output of the program

Class Demo

Class Demo2

Class Demo2

Inherited, Overridden & Specialized properties :-



Inherited properties :- Inherited properties are such

properties which are inherited from the parent class into the child class and are used in the child class without modification. as it is.

Advantage :-

* code reusability.

Overridden properties :- Overridden properties are such.

properties which are inherited from the parent class but are modified in the child class in order to meet the needs of the child class.

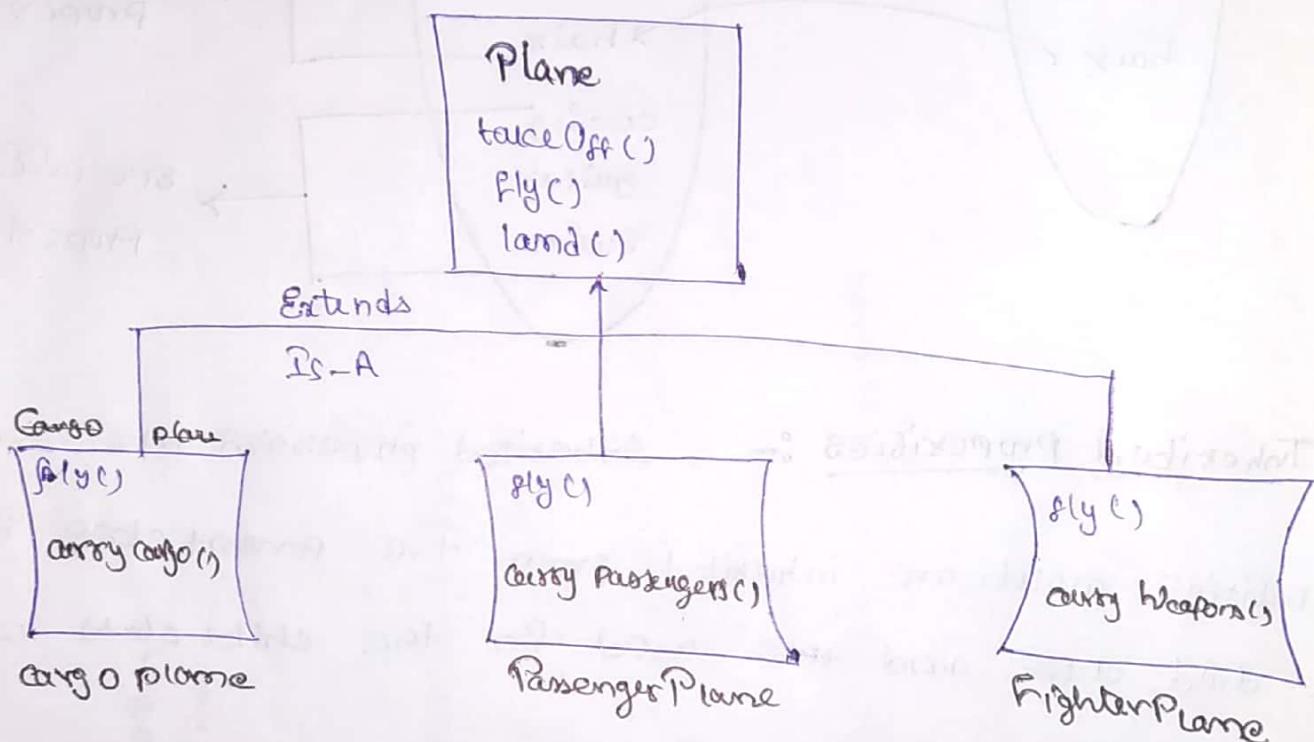
Adv :- They satisfy the needs of child class (of) project.

Specialized properties :- Specialized properties are such properties which are present only in the child class and they are not inherited.

Adv :- It enriches the features of the child class.

Inheritance - Program - 1

UML - diagram :-



Class Plane

```
{  
    public void takeOff()  
    {  
        System.out.println("plane is taking off");  
    }  
  
    public void fly()  
    {  
        System.out.println("plane is flying");  
    }  
  
    public void land()  
    {  
        System.out.println("plane is landing");  
    }  
}
```

Class CargoPlane extends Plane

```
{  
    public void fly()  
    {  
        System.out.println("Cargo plane is flying at lower heights");  
    }  
  
    public void carryCargo()  
    {  
        System.out.println("Cargo plane is carrying goods");  
    }  
}
```

Class PassengerPlane extends Plane

```
{  
    public void fly()  
    {  
        System.out.println("PassengerPlane is flying at medium heights");  
    }  
  
    public void carryPassengers()  
    {  
        System.out.println("Passenger plane is carrying passengers");  
    }  
}
```

{ Class FighterPlane extends Plane

{ Public void fly()

{ S.o.p("Fighterplane is flying at great heights");

}

Public void carryWeapons()

{ S.o.p("Fighterplane is carrying Weapons");

}

}

Class Launch

{

public static void main (String args [])

{

CargoPlane CP = new CargoPlane();

PassengerPlane PP = new PassengerPlane();

FighterPlane FP = new FighterPlane();

CP.takeOff();

CP.fly();

CP.land();

CP.carryCargo();

S.o.println();

PP.takeOff();

PP.fly();

PP.land();

PP.carryPassengers();

S.o.println();

FP.takeOff();

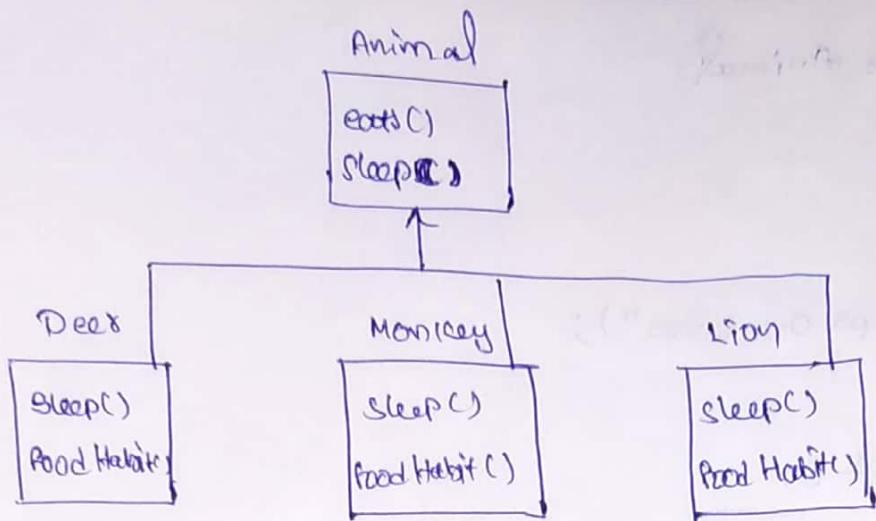
FP.fly();

FP.land();

FP.carryWeapons();

}

}



Class Animal

```

{
    public void eats() {
        {
            S.O.P ("Animal is eating");
        }
    }

    public void sleep() {
        {
            S.O.P ("Animal is sleeping");
        }
    }
}
  
```

Class Deer extends Animal

```

{
    public void sleep() {
        {
            S.O.P ("Deer sleeps on grass");
        }
    }

    public void foodHabit() {
        {
            S.O.P ("Deer is Herbivorous");
        }
    }
}
  
```

{ class Monkey extends Animal

Public void sleep()

{ S.O.P ("Monkey sleeps on trees"); }
}

```
public void FoodHabit()
```

S.O.P.C. mobility is omnivorous);

class Lion extends Animal

۱

public void sleep()

۸

S.O.P ("Lion sleeps on cawcs");

m.eats();

m: sleep();

m. food Habit ():

l. eats();

I sleep(s);

1. Food Habits:

Class Law Ch

```
{  
    } public void main(String args[]) {
```

```
Dear d = new Deer();
```

```
Monkey m = new Monkey();
```

```
Lion l = new Lion();
```

d. eatS();

d. sleeps();

d. Food Habit () :

Method Overriding :-

It is a process of providing a different body to the inherited method in child class. as shown in below.

Class Demo

```
{ public void disp()
{ System.out.println("ABC for Technology Training"); }
}
```

Class Demo2 extends Demo

```
{ public void disp()
{ System.out.println("ABC for Java and Testing"); }
}
```

class launch

```
{ public static void main(String args[])
{ Demo2 d2 = new Demo2();
d2.disp(); }
}
```

O/P :- ABC for Java and Testing

Rule-1

During method overriding the overridden method

in the child class should be given same access privilege (or) higher privilege compare to parent class method. Weaker access privileges can't be assigned to the overridden method.

class Demo

{

 public void fun()

{

 System.out.println("BTM");

}

Class Demo2 extends Demo

{

 void fun() → Exxox

{

 System.out.println("Vijayanagar");

}

class Launch

{

 public static void main(String args[])

{

 Demo2 d2 = new Demo2();

 d2.fun();

}

Rule-2

The return type of the overridden method

should be same as parent class method. as

Shown in below.

Class Demo

```
{  
    public void fun()  
    {  
        System.out.println("BTM");  
    }  
}
```

Class Demo2 extends Demo

```
{  
    public int func() → Error  
    {  
        System.out.println("vijaynagar");  
        return 1;  
    }  
}
```

Class Launch

```
{  
    public void string args()  
    {  
        Demo2 d2 = new Demo2();  
        d2.fun();  
    }  
}
```

Rule - 3 :-

The overridden method in the child class can have a different return type when compared with parent class method if and only if 'IS-A' relationship exists between the return types.

Such return types which have 'IS-A' relationship between them are called as Co-variant return types.

* Legal Case :-

Program ↓

* Illegal Case :-

Program ↓

class Animal // Legal

{
}

class Tiger extends Animal

{
}

class Demo

{

public Animal func()

{
 S.O.P("BTM");

 Animal a = new Animal();

 return a;

}
}

class Demo2 extends Demo

{

public Tiger func()

{
 S.O.P("Vijayanagar");

 Tiger t = new Tiger();

 return t;

}
}

class Launch

{ psvm (String args[])

{

 Demo2 d2 = new Demo2();

 d2.fun();

}
}

O/P :- Vijayanagar

class Animal {} // Illegal

class Tiger extends Animal

{
}

class Demo

{

 public Tiger func();

{

 S.O.P("BTM");

 Tiger t = new Tiger();

 return t;

}
}

class Demo2 extends Demo

{

 public Animal func() → Error

{

 S.O.P("Vijayanagar");

 Animal a = new Animal();

 return a;

}
}

class Launch

{ psvm (String args[])

{

 Demo2 d2 = new Demo2();

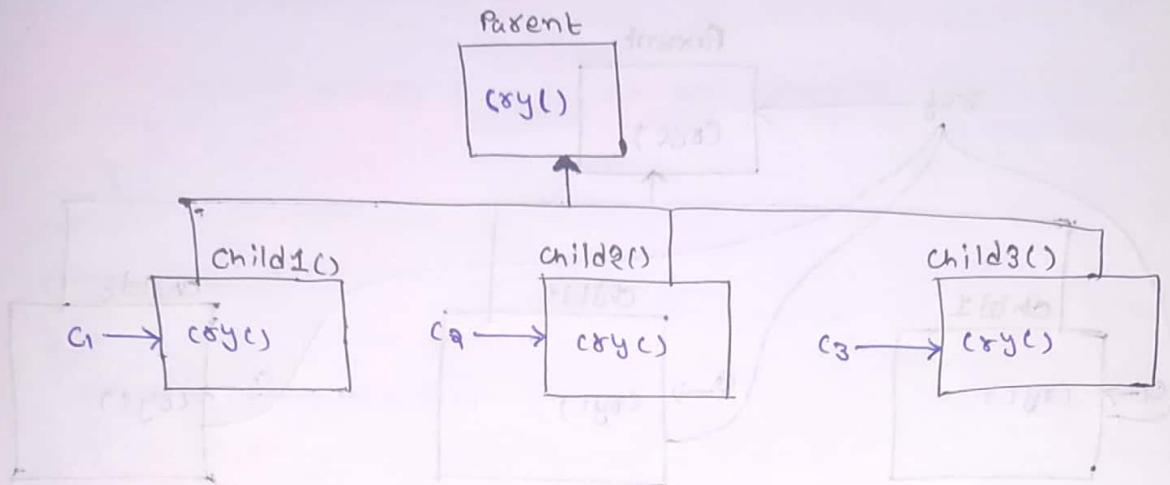
 d2.fun();

}
}

O/P :- Error

11109119

Non-Polyomorphic Version



class Parent

۱

```
public void cry ()  
{  
    System.out.println("Parent doesn't cry");  
}
```

Class Child I extend Parent

۱

```
public void copy()
```

۹۲

child extends Parent

۲

```
public void copy()
```

8

E
so.p("child2 is saying at
medium volume");

33

class Child3 extends Parent

۹

{ public void copy()

{

S.O.P ("child3 is crying at high volume");

۳۵

Class Launch

2

sum (String args[])

۱

child c₁ = new Child1();

```
Child2 c2 = new Child2();
```

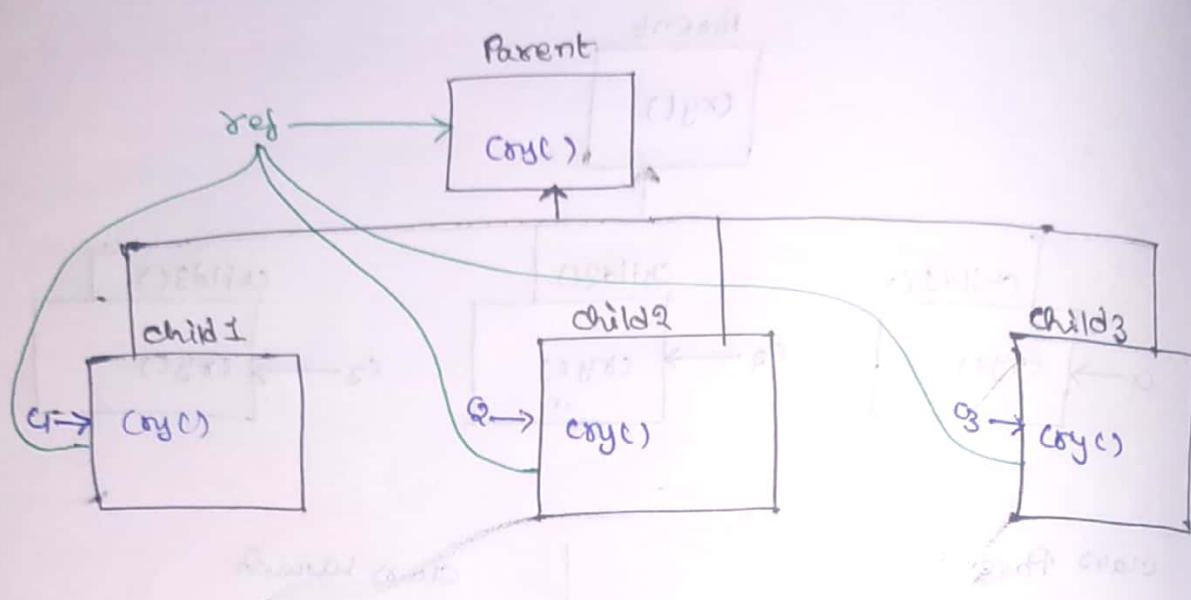
```
Child c3 = new Child3();
```

cl. (myc); 1:1

C₂. cry(); 1:1

C₃. C₆S(1); 1:1

Poly morphic Version



Class Parent

```
{  
    public void cry()  
    {  
        System.out.println("Parent doesn't cry");  
    }  
}
```

class child1 extends Parent

```
{  
    public void cry()  
    {  
        System.out.println("child1 is crying at low volume");  
    }  
}
```

class child2 extends Parent

```
{  
    public void cry()  
    {  
        System.out.println("child2 is crying at medium volume");  
    }  
}
```

class Child3 extends Parent

```
{  
    public void cry()  
    {
```

```
        System.out.println("Child3 is crying at high volume");
```

```
}
```

class Launch

```
{
```

```
    public static void main(String args[])
```

```
{
```

```
    Child1 c1 = new Child1();
```

```
    Child2 c2 = new Child2();
```

```
    Child3 c3 = new Child3();
```

```
    Parent ref;
```

```
    ref = c1;
```

```
    ref.cry();
```

```
    ref = c2;
```

```
    ref.cry();
```

```
    ref = c3;
```

```
    ref.cry();
```

```
}
```

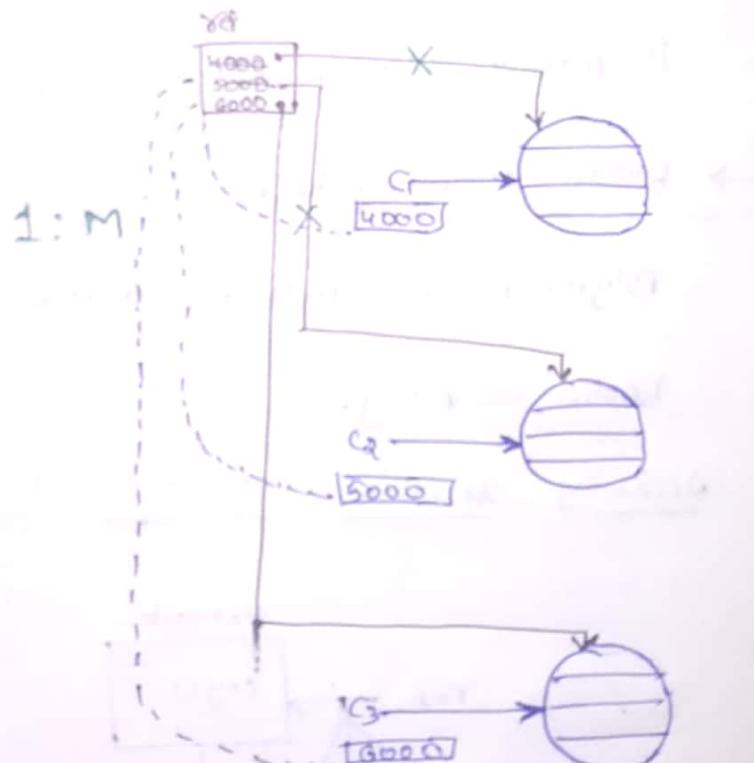
```
}
```

O/P:-

child1 is crying at low volume

child2 is crying at medium volume

child3 is crying at high volume



Tight coupling :-

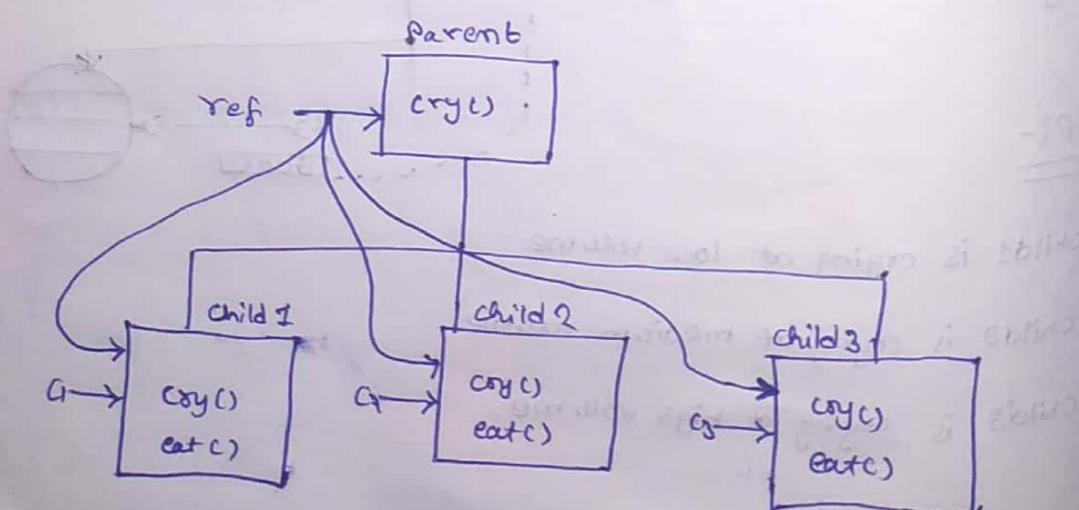
If a child type reference is created for a child type object. In other words if child type reference is assigned to child type object then it is called as tight coupling. In tight coupling the type of reference and type of object are same.

Loose coupling :-

Creating a parent type reference for child objects is in other words assigning parent type reference to child objects is called loose coupling. In loose coupling the type of reference is parent class, the type of object is child class.

→ loose coupling helps us in achieving polymorphism, polymorphism can't be achieved without using loose coupling.

Accessing specialized properties using Parent reference



class Parent

```
{ public void cry()
```

```
{ System.out.println("Parent doesn't
```

```
cry"); }
```

```
}
```

class Launch

```
{ public void child1() {
```

```
System.out.println("child1 args[" +
```

```
args[0] + "]"); }
```

```
{ public void child2() {
```

```
System.out.println("child2 args[" +
```

```
args[1] + "]"); }
```

```
{ public void child3() {
```

```
System.out.println("child3 args[" +
```

```
args[2] + "]"); }
```

class Child1 extends Parent

```
{
```

```
public void cry()
```

```
{
```

```
S.O.P("child1 is crying at low  
volume");
```

```
}
```

```
public void eat()
```

```
{
```

```
S.O.P("child1 eats less food");
```

```
}
```

```
}
```

Parent ref;

```
ref = c1;
```

```
ref.cry();
```

```
(Child1)(ref).eat();
```

```
ref = c2;
```

```
ref.cry();
```

```
(Child2)(ref).eat();
```

```
ref = c3;
```

```
ref.cry();
```

```
(Child3)(ref).eat();
```

class Child2 extends Parent

```
{
```

```
public void cry()
```

```
{
```

```
S.O.P("child2 is crying at
```

```
medium volume");
```

```
}
```

```
public void eat()
```

```
{
```

```
S.O.P("child2 eats sufficient
```

```
food");
```

```
}
```

class Child3 extends Parent

```
{ public void cry()
```

```
{
```

```
S.O.P("child3 is crying
```

```
at high volume");
```

```
}
```

```
public void eat()
```

```
{
```

```
S.O.P("child3 eats more
```

```
food");
```

→ Using loose coupling we can achieve polymorphism. Using Parent type reference loose coupling can be achieved. But however Parent type reference can access only overridden and inherited properties of the child class, It can't access specialized properties of child class. Hence down casting has to be performed.

Down Casting :-

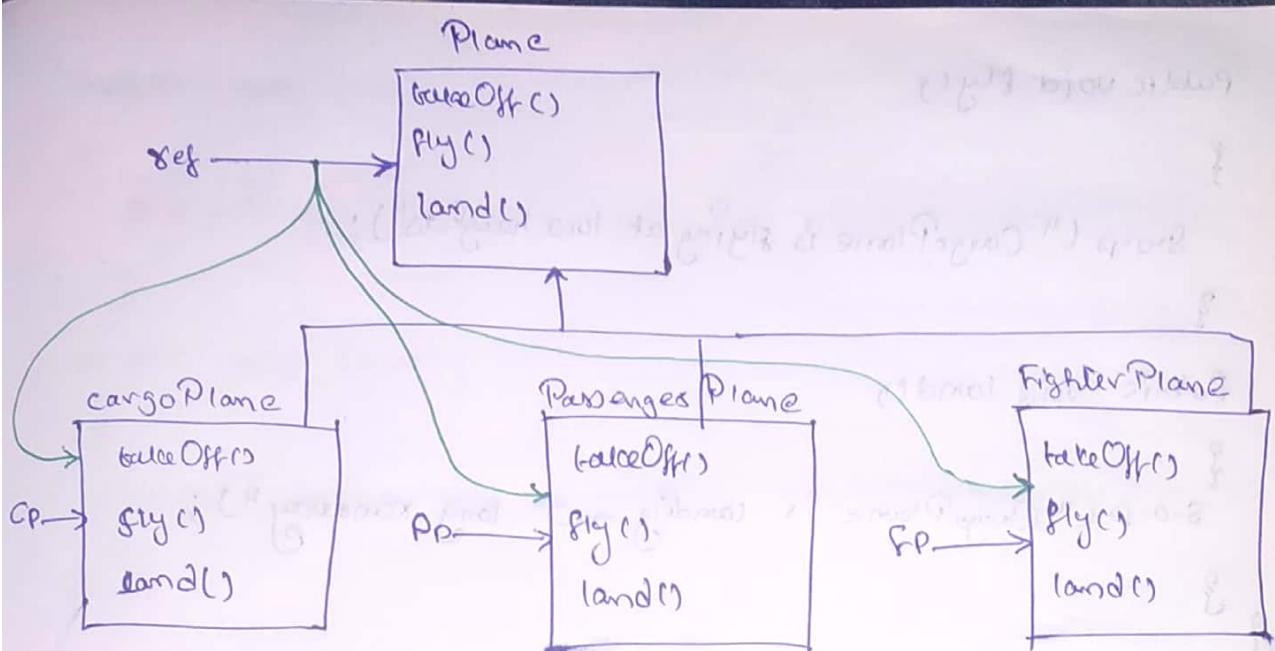
Down Casting is a process where parent type reference is temporarily converted into child type in order to access specialized properties of child class.

→ Note:-

All the user defined methods in the user defined class will be treated as specialized by object class. Becoz object class is the parent of all the classes. Hence in order to access user defined methods using object reference down casting is the only option available.

```
Class Demo extends Object Class Launch
{
    void fun()
    {
        System.out.println("Hello");
    }
}

class Demo
{
    public static void main(String args[])
    {
        Demo d = new Demo();
        Object ref = d;
        ((Demo)(ref)).fun();
    }
}
```



Class Plane

```
{
    public void takeOff()
}
```

```
{
    System.out.println("Plane is taking off");
}
```

```
public void fly()
```

```
{
    System.out.println("Plane is flying");
}
```

```
public void land()
```

```
{
    System.out.println("Plane is landing");
}
```

class CargoPlane extends Plane

```
{
    public void takeOff()
```

```
{
    System.out.println("CargoPlane is taking off from long
                        runway");
}
```

```
public void fly()
```

1

S.O.P ("CargoPlane is flying at low heights");

3

Public void land()

8

S.O.P ("Cargo Plane is landing on long runway");

3

十一

Class PassengerPlane extends Plane

۸

public void takeOff()

P

S.O.P ("Passenger Plane is taking off from medium runway")

۹

public void Fly()

۲۰

S.O.P ("Passenger Plane is flying at medium height");

۳

public void land()

5

S.O.P.C "Passenger Plane" is landing from medium runway

3 5

Class FighterPlane extends Plane

۲

Public void takeOff()

2

S.O.P ("Fighter Plane is taking off from short runway")

```
public void fly()
```

```
{
```

```
System.out.println("Fighter Plane is flying at great height");
```

```
}
```

```
public void land()
```

```
{
```

```
System.out.println("Fighter Plane is landing on short runway");
```

```
}
```

```
class Launch
```

```
{
```

```
public static void main (String args [] )
```

```
{
```

```
CargoPlane cp = new CargoPlane();
```

```
PassengerPlane pp = new PassengerPlane();
```

```
FighterPlane fp = new FighterPlane();
```

```
Plane ref ;
```

```
ref = cp;
```

```
ref.takeOff();
```

```
ref.fly();
```

```
ref.land();
```

```
ref = pp;
```

```
ref.takeOff();
```

```
ref.fly();
```

```
ref.land();
```

```
ref = fp;
```

```
ref.takeOff();
```

```
ref.fly();
```

```
ref.land();
```

```
}
```

Advantage of polymorphism :-

Class Home

{ } == }

Closes CargoPlane extends Plane

20

class PassengerPlane extends Plane

四
四

class FighterPlane extends Plane

卷二

Closes Airport

2

Public void Permit (Plane ref)

def. balanceOff():

ref. Fly (1);

ref. last();

۳

class launch

۲۷

$P \propto n^m C \text{String} \arg \{z\}$

{

```
CargoPlane cp = new CargoPlane();  
PassengerPlane pp = new PassengerPlane();  
FighterPlane fp = new FighterPlane();  
  
Airport a = new Airport();  
a.permit(cp);  
a.permit(pp);  
a.permit(fp);
```

{ }

CargoPlane CP = new CargoPlane();

PassengerPlane PP = new PassengerPlane();

FighterPlane FP = new FighterPlane();

Airport a = new Airport();

a.permit(CP);

a.permit(PP);

a.Permit(FP);

}

Up Casting :- Up Casting is same as Loose Coupling.

Loose Coupling :- Creating Parent reference to child ~~object~~

Type object.

Tight Coupling :- Creating child type reference to

child type object.

Down Casting :- Temporarily converting parent type

reference to child type in order to access the

specialized methods and properties of the child

class.

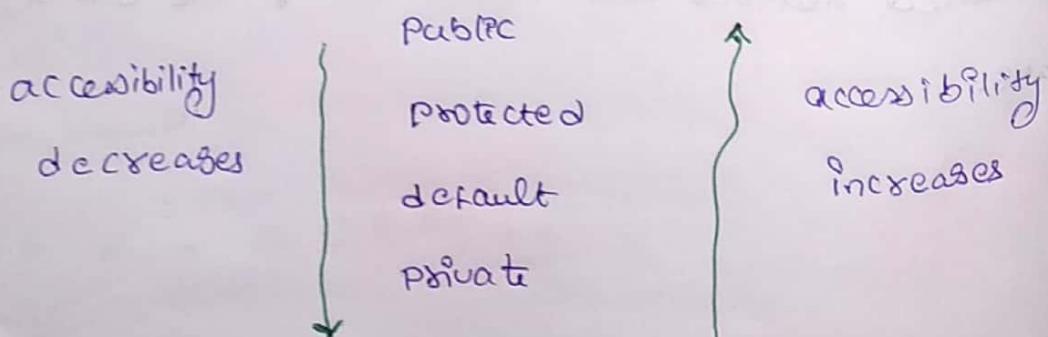
Access modifiers | Access specifiers

Public :- Public data members and methods are accessible throughout the project.

Protected :- Protected datamembers and methods are accessible throughout the package and also in the child classes of different package.

Default :- Default datamembers have package level access which means they are accessible only within the package where they are created.

Private :- Private data members and methods are accessible only within the class where they are created.



ABC

P₁

```
class Demo
{
    public int a=10; ✓
    protected int b=20; ✓
    int c = 30; ✓
    private int d=40; ✓
}
```

```
class Demo2 extends Demo
```

```
{
    ✓✓✓
    a b c d x
}
```

```
class Demo3
```

```
{
    ✓✓✓
    abc d x
}
```

P₂

```
class Demo4 extends Demo
{
    ✓✓✓
    a b c d x
}
```

```
class Demo5
```

```
{
    ✓xxx
    a b c d
}
```

Delegation model :-

```
class Worker
{
    public void dusting()
    {
        s.o.p("dusting activity in progress");
    }
}
```

```
public void cleaning()
```

```
{
    s.o.p("cleaning activity in progress");
}
```

```
public void washing()
```

```
{
    s.o.p("washing activity in progress");
}
```

```
}
```

Class Supervisor

```
{  
    Worker w = new Worker();  
  
    public void dusting()  
    {  
        w.dusting();  
    }  
  
    public void cleaning()  
    {  
        w.cleaning();  
    }  
  
    public void washing()  
    {  
        w.washing();  
    }  
}
```

Class Owner

```
{  
    String args[];  
  
    public void main(String args[])  
    {  
        Supervisor s = new Supervisor();  
  
        s.dusting();  
        s.cleaning();  
        s.washing();  
    }  
}
```

Output

Dusting activity is in progress.

Cleaning activity is in progress.

Washing activity is in progress.

Notes

In order to access the parent class methods and properties
 In the child class Super keyword is used.

Class Demo

{

int i=9;

public void disp()

{

System.out.println("Hello");

}

}

class Demo2 extends Demo

{

int i=99;

public void disp()

{

System.out.println(i);

super.disp();

}

public void disp2()

{

disp();

super.disp();

}

}

class Launch

{

public static void main(String args[])

{

Demo2 d2 = new Demo2();

d2.disp2();

}

}

(Output) 99

9

Hello

Aggregation and Composition :-

In this real world we not only have 'Is-a' relationship, we also have 'has-a' relationship.

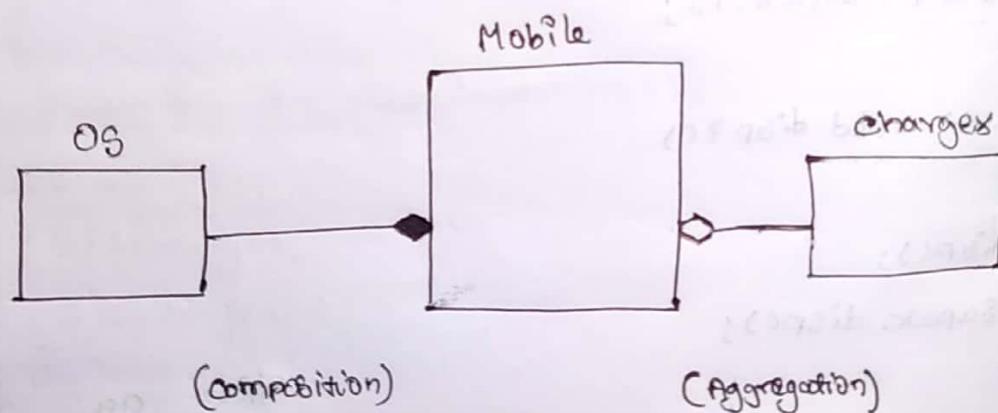
They are two types of "has-a" relationships.

i) Tight-Bound 'has-a' relationship.

ii) Loose-Bound 'has-a' relationship.

→ Tight Bound 'has-a' relationship is only known at Composition.

→ Loose Bound 'has-a' relationship is only known at Aggregation.



class OS

{

private String name;

private int size;

public OS (String name, int size)

{

this.name = name;

this.size = size;

{

public String getName()

{

return name;

}

public int getSize()

{

return size;

}

{

class Charger

{

private String brand;

private float voltage;

public Charger (String brand, float voltage)

{

this.brand = brand;

this.voltage = voltage;

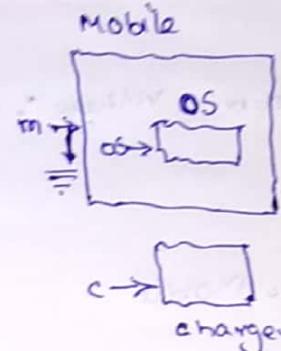
{

public String getBrand()

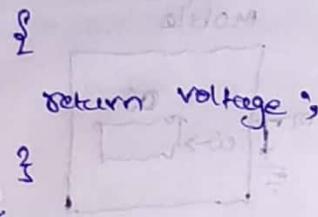
{

return brand;

{



public float getVoltage()



class Mobile

reports

{

OS os = new OS("Android", 512);

public void have(Charger c)

{

S.O.P(c.getBrand());

S.O.P(c.getVoltage());

}

class Launch

{

System.out.println(m.toString(args[1]))

{

Mobile m = new Mobile();

S.O.P(m.os.getName());

S.O.P(m.os.getSize());

Charger c = new Charger("Nokia", 2.8f)

m.have(c);

S.O.P(c.getBrand());

S.O.P(c.getVoltage());

}/

20 marks

Answer given below

part 3 marks

(part 3) (answering) 20 marks

if m == null, print

else // S.O.P(m.os.getName());

// S.O.P(m.os.getSize());

// m. has A c;

(Charger c, getBrand());

S.O.P(c.getBrand());

S.O.P(c.getVoltage());

}

(part 4) 20 marks

System.out.println(m)

getBrand()

brand prints same

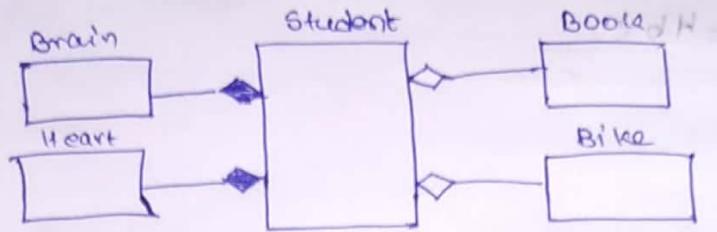
regular tools, nothing

brand & brand will

negative = regular will

Charger is not present

Brand owners



class Brain

{

private string name; ~~private int size;~~

private int size;

public Brain (string name, int size)

{

this.name = name;

this.size = size;

}

public string getBook()

{

return @book;

}

public int getSize()

{

return size;

}

}

class Heart

{

private int hb;

private int size;

public Heart (int hb, int size)

{

this.hb = hb;

this.size = size;

}

public int getHb()

{

return hb;

}

public int getSize()

{

return size;

}

class Book

{

private String name;

private int cost;

public Book (String name, int cost)

{

this.name = name;

this.cost = cost;

}

public String getName()

{

return name;

}

public int getSize()

{

return size;

}

class Bike

{

private String brand;

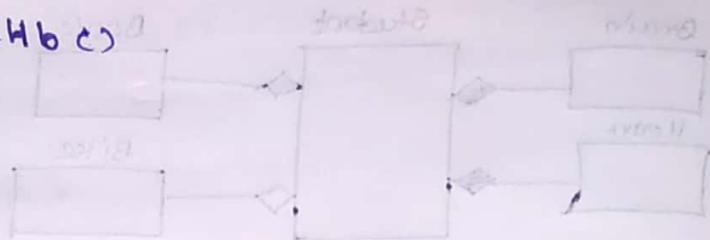
private float mileage;

public Bike (String name, float mileage)

{ this.brand = brand;

this.mileage = mileage;

}



```
public String getBrand()
{
    return board;
}

public float getMileage()
{
    return mileage;
}
```

```
class Student
{
    String name;
}
```

```
Brain B = new Brain("Grey", 150);
```

```
Heart H = new Heart(72, 250);
```

```
public void hasA(Book B)
```

```
{
    System.out.println(B.getName());
    System.out.println(B.getCost());
}
```

```
public void hasA(Bike B)
{
    System.out.println(B.getBrand());
    System.out.println(B.getMileage());
}
```

```
class Launch
{
    String args[];
}

public void run(String args[])
{
    Student S = new Student();
}
```

s.o.p(S.B.getColour());

s.o.p(S.B.getSize());

s.o.p(S.H.getHb());

s.o.p(S.H.getSize());

Book Bo = new Book("Harry Potter", 550);

S.hasA(Bo);

s.o.p(Bo.getName());

s.o.p(Bo.getCost());

Bike Bi = new Bike("Duke", 32.5);

S.hasA(Bi);

s.o.p(Bi.getBrand());

s.o.p(Bi.getMileage());

small;

//s.o.p(S.B.getColor());

//s.o.p(S.B.getSize());

//s.o.p(S.H.getHb());

//s.o.p(S.H.getSize());

/*S.hasA(Bo);

/*S.hasA(Bi);

s.o.p(Bo.getName());

s.o.p(Bo.getCost());

s.o.p(Bi.getBrand());

s.o.p(Bi.getMileage());

Output :-

Grey
150

72

550

Harry Potter

550

Harry Potter

550

Duke

32.5

Duke

32.5

Harry Potter

550

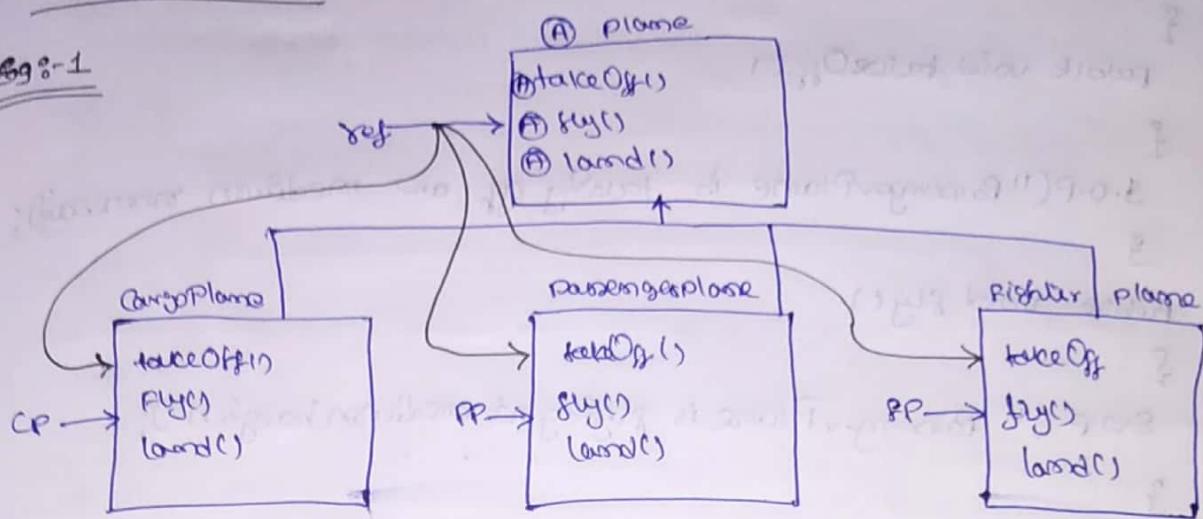
Duke

32.5

}

ABSTRACTION :-

Eg :- 1



abstract class Plane

{

 abstract public void takeOff();

 abstract public void fly();

 abstract public void land();

}

class CargoPlane extends Plane

{

 public void takeOff()

{

 System.out.println("CargoPlane is taking off from long runway");

}

 public void fly()

{

 System.out.println("CargoPlane is flying at low heights");

}

 public void land()

{

 System.out.println("CargoPlane is landing on long runway");

}

class PassengerPlane extends Plane

{

 public void takeOff()

{

 S.O.P("PassengerPlane is taking off on medium runway");

}

 public void fly()

{

 S.O.P("PassengerPlane is flying at medium heights");

}

 public void land()

{

 S.O.P("PassengerPlane is landing on medium runway");

}

class FighterPlane extends Plane

{

 public void takeOff()

{

 S.O.P("FighterPlane is taking off at short runway");

}

 public void fly()

{

 S.O.P("FighterPlane is flying at great heights");

}

 public void land()

{

 S.O.P("FighterPlane is landing on short runway");

}

 {("written for no printed diagram")}

class Airport

```
{  
    public void permit (Plane ref)  
    {  
        ref.takeOff();  
        ref.fly();  
        ref.land();  
    }  
}
```

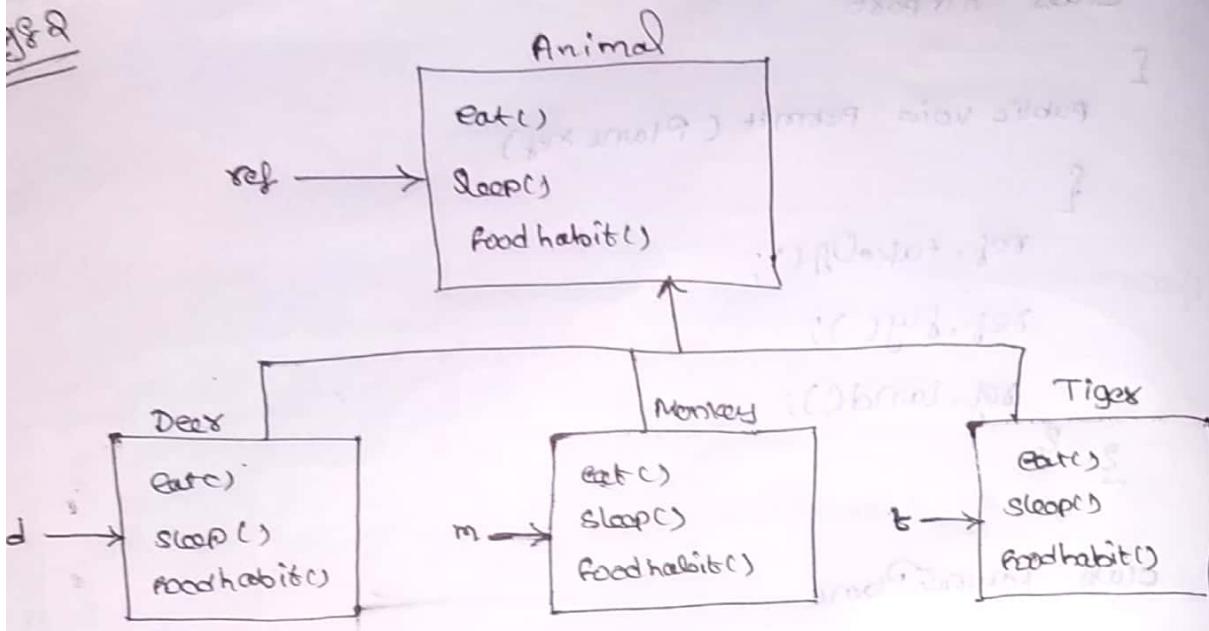
class LaunchPlane

```
{  
    public static void main (String args [])  
    {  
        CargoPlane CP = new CargoPlane ();  
        PassengerPlane PP = new PassengerPlane ();  
        FighterPlane FP = new FighterPlane ();  
  
        Airport a = new Airport ();  
  
        a.permit (CP);  
        a.permit (PP);  
        a.permit (FP);  
    }  
}
```

Output &
CargoPlane is taking off at long runway.
CargoPlane is flying at low heights.
CargoPlane is landing on long runway.

PassengerPlane is taking off at Medium runway.
PassengerPlane is flying at medium heights.
Passenger Plane is landing on medium runway.

FighterPlane is taking off at short runway.
Fighter Plane is flying at great heights.
FighterPlane is landing on short runway.



abstract class Animal

{
 abstract public void eat();
 abstract public void sleep();
 abstract public void foodhabit();

}
class Deer extends Animal

{
 public void eat()
 {
 System.out.println("Deer eats grass");
 }

}
public void sleep()

{
 System.out.println("Deer sleeps on grass");
}

}
public void foodhabit()

{
 System.out.println("Deer is herbivorous");
}

}
}

class Monkey extends Animal

{
public void eat()
{

s.o.p("Monkey eats biscuits");

}
public void sleep()

s.o.p("Monkey sleeps on trees");

}
public void foodHabit()

{
s.o.p("Monkey is Omnivorous");

}
class Tiger extends Animal

{
public void eat()
{

s.o.p("Tiger eats flesh");

}
public void sleep()

{
s.o.p("Tiger sleeps on lava");

}
public void foodHabit()

{
s.o.p("Tiger is Carnivorous");

class Forest

{
public void permit(Animal ref)

{
ref.eat();
ref.sleep();

```
ref. foodHabit();
```

```
{  
}
```

```
class Animal {
```

```
{
```

```
    public void eat(String args[]) {
```

```
{
```

```
    Deer d = new Deer();
```

```
    Monkey m = new Monkey();
```

```
    Tiger t = new Tiger();
```

```
    Forest f = new Forest();
```

```
    f.permit(d);
```

```
    f.permit(m);
```

```
    f.permit(t);
```

```
{  
}
```

Output

```
Deer eats grass.
```

```
Deer sleeps on grass.
```

```
Deer is herbivorous.
```

```
Monkey eats fruits.
```

```
Monkey sleeps on trees.
```

```
Monkey is Omnivorous.
```

```
Tiger eats flesh.
```

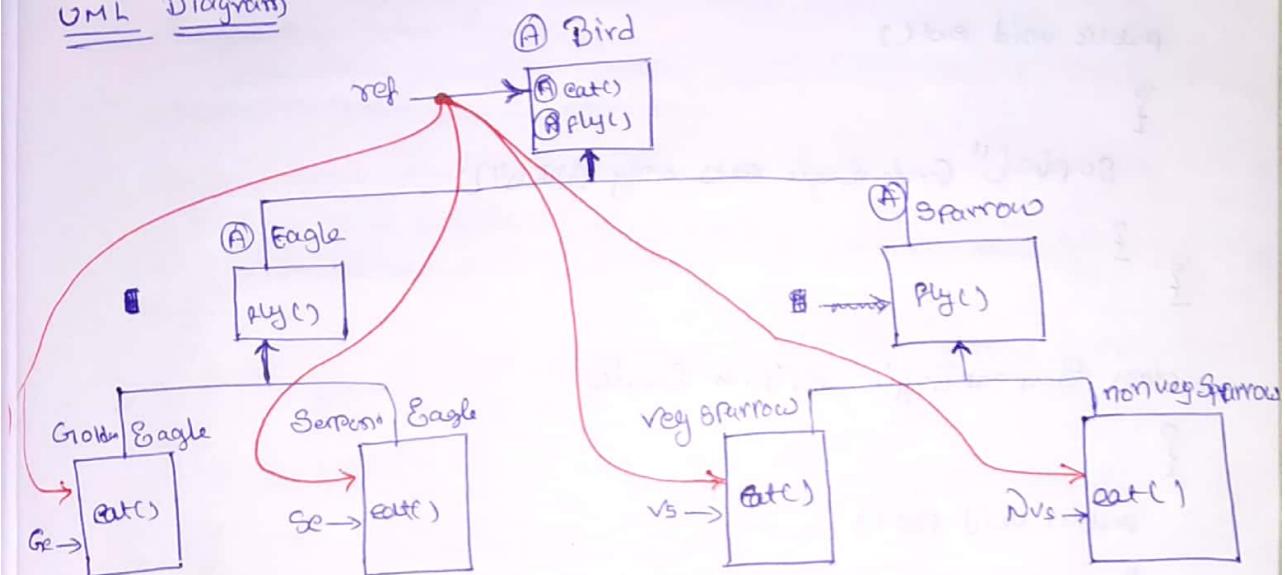
```
Tiger sleeps on caves.
```

```
Tiger is Carnivorous.
```

Example - 3

unified modeling language

UML Diagram



① class Bird

```

{
    abstract public void eat();
    abstract public void fly();
}
  
```

② class Eagle extends Bird

```

{
    public void fly() {
        System.out.println("Eagles fly at great heights");
    }
}
  
```

③ class Sparrow extends Bird

```

{
    public void fly() {
        System.out.println("Sparrow flies at medium height");
    }
}
  
```

class GoldenEagle extends Eagle

{

 public void eat()

{

 System.out.println("GoldenEagle eats only fish!");

}

class SerpentEagle extends Eagle

{

 public void eat()

{

 System.out.println("SerpentEagle eats only snakes!");

}

class VegSparrow extends Sparrow

{

 public void eat()

{

 System.out.println("Veg Sparrow eats only grains!");

}

}

class NonVegSparrow extends Sparrow

{

 public void eat()

{

 System.out.println("NonVeg Sparrow eats only worms!");

}

}

class Sky

{

Eagle e = new Eagle();

Sparrow s = new Sparrow();

public void freedom(Bird ref)

{

ref. fly();

ref. eat();

}

class LaunchBird

{

p s v m (String args[])

{

GoldenEagle Ge = new GoldenEagle();

SerpentEagle Se = new SerpentEagle();

Veg Sparrow Vs = new VegSparrow();

NonVegSparrow Nvs = new NonVegSparrow();

Sky s = new Sky();

s.freedom(Ge);

s.freedom(Se);

s.freedom(Vs);

s.freedom(Nvs);

}

Abstract

Abstract class is a class which can't be instantiated.

- An Abstract class can have only abstract methods.
- An Abstract class can have only concrete methods.
- An Abstract class can have both abstract and concrete methods.

Abstract Method :-

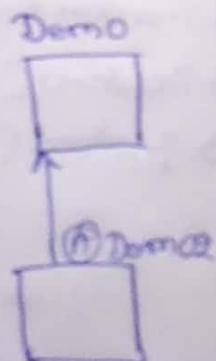
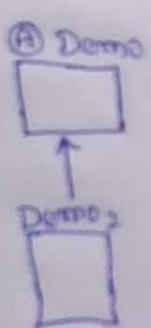
Abstract method is a method which has only method signature. It doesn't have method body.

Concrete method:

A Concrete method has both method signature and method body.

Note:-

- Abstract class can be inherited by both abstract and non-abstract classes.
- Abstract class can inherit both abstract and non-abstract class.



Note :-

* There is only one difference b/w Abstract class and Non-Abstract class which is object of the non-Abstract class can be created. Object of the Abstract class can't be created. Hence in order to avoid the object creation of a class abstract key word can be used. *

final key word :-

- The final keyword can be apply on a variable, method and class.
- If final keyword is applied on a method it can be inherited but it can't be overridden. In other words final methods can't be overridden.

Program:-

```
class Demo
{
    final public void add()
    {
        System.out.println ("Inside Demo class");
    }
}

class Demo2 extends Demo
{
    final public void add() //---> Error
    {
        System.out.println ("Inside Demo2 class");
    }
}
```

class Launch

```
{  
    public static void main (String [] args)  
    {  
        Demo2 d2 = new Demo2();  
        d2.add();  
    }  
}
```

Note:-

→ The data members and methods present in final class can't be inherited. (or) In other words a final class can't be extended.

Program:-

final class Demo

```
{  
    public void disp()  
    {  
        System.out.println("Welcome to ABC");  
    }  
}
```

class Demo2 extends Demo // Error

```
{  
    public void disp()  
    {  
        System.out.println("This is BTM Control");  
    }  
}
```

class Launch

{

 ps v m c String args[])

{

 Demo d2 = new Demo();

 d2.disp();

}

→ A final variable can be inherited but however it can't be assigned a different value either in child class or same class where it is initialized.

class Demo

{

 final int i=50;

 public void disp()

{

 i=100; → Error

 s.o.p(i);

}

}

class Demo2 extends Demo

{

}

class Launch

{ ps v m c String args[])

{

 Demo2 d2 = new Demo2();

 d2.disp();

}

}

Valid Signatures of main method :-

public static void main (String args [])

public static void main (String [] args)

static public void main (String args [])

public static void main (String... args)

Versions of Java :-

Java 1.0

→ OAK

→ Released on January 23, 1996

Java 1.1

→ Released on February 19, 1997.

Java 1.2

→ Playground

→ Released on December 8, 1998.

Java 1.3

→ Kestrel

→ Released on May 8, 2000.

Java 1.4

→ Merlin

→ Released on February 6, 2002.

Java 5

→ Tigris

→ Released on September 30, 2004.

Java 6

→ Mustang

→ Released on December 11, 2006.

Java 7

→ Dolphin

→ Released on July 28, 2011.

Java 8

→ Spider

→ Released on March 18, 2014.

Java 9

→ Released on 21 September, 2017.

Java 10

→ Released on 20 March, 2018.

Java 11

→ Released on 25 September, 2018.

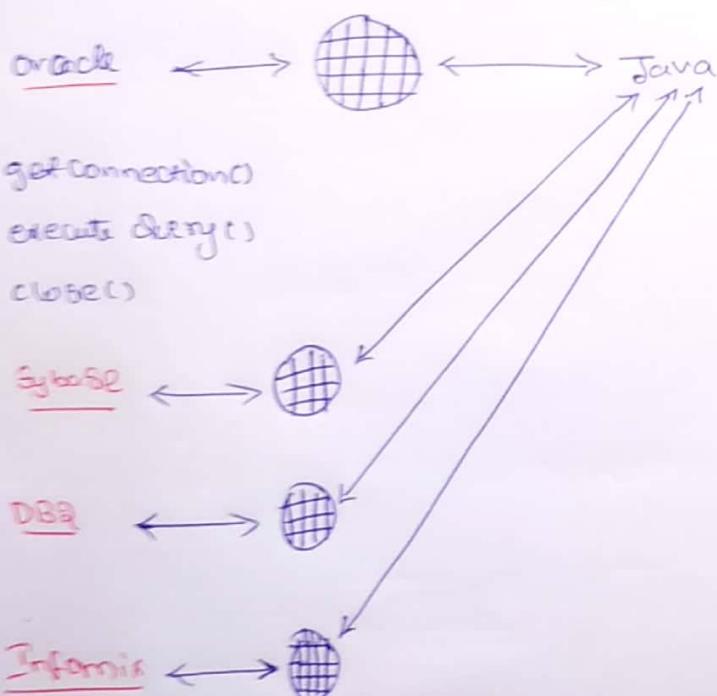
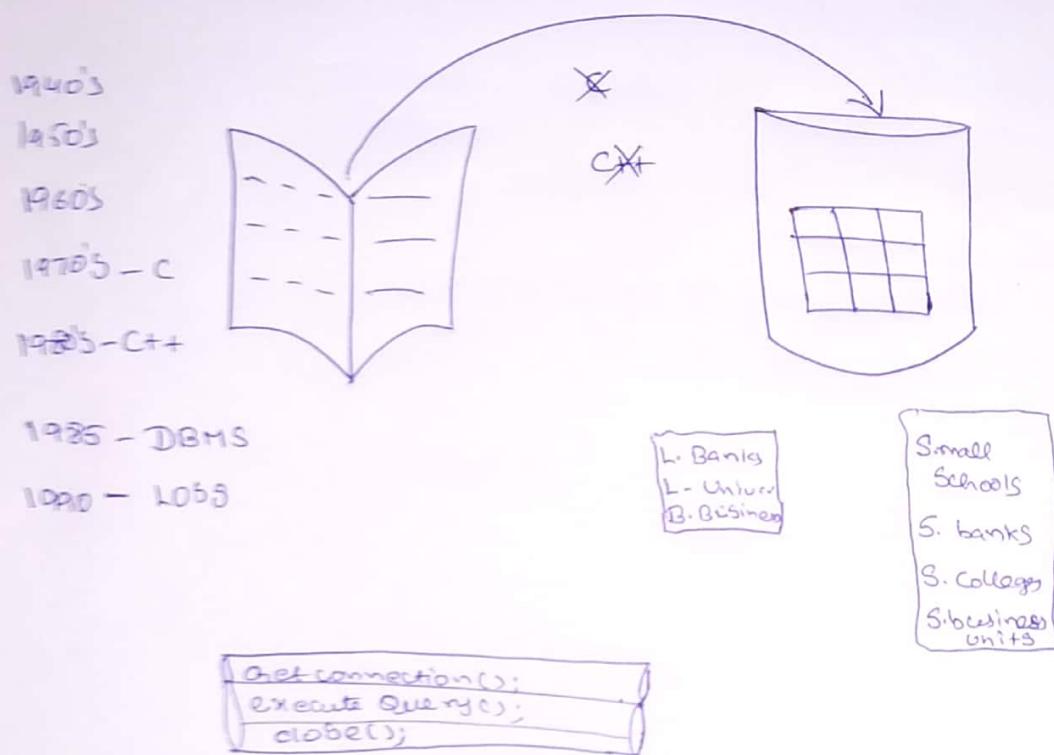
Java 12

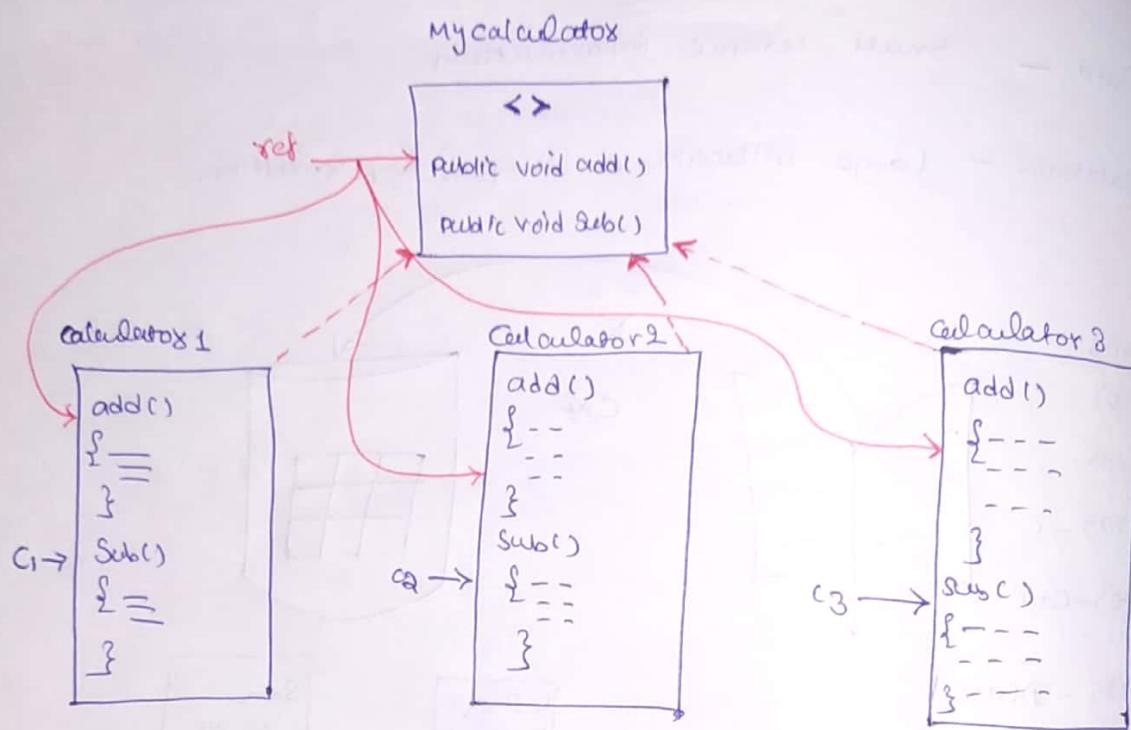
→ Released on 19 March, 2019.

Interfaces

Data — Small useful information.

Database — Large collection of useful information.





Rule - 1

Interfaces are used to achieve Standardization.

Rule - 2

One interface can have any number of Implementations.

Rule - 3

Using Interface we can achieve loose coupling.

Rule - 4

Using Interface we can achieve Polymorphism.

(Ans) Interfaces promote polymorphism.

Program

```
import java.util.Scanner;  
interface MyCalculator  
{  
    public void add();  
    public void sub();  
}
```

```
class Calculator1 implements MyCalculator
```

```
{  
    public void add()  
    {
```

```
        int a=10;
```

```
        int b=20;
```

```
        int c=a+b;
```

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

```
public void sub()  
{  
    int a=10;  
    int b=20;  
    int c=a-b;  
    System.out.println(c);  
}
```

```
}  
}
```

```
class Calculator2 implements MyCalculator
```

```
{  
    public void add()  
    {
```

```
        int a;  
        int b;
```

```
        Scanner scan = new Scanner(System.in);  
        System.out.println("Enter the first number");  
        a = scan.nextInt();  
    }
```

```
    public void sub()  
    {  
        int a=10;  
        int b=20;  
        int c=a-b;  
        System.out.println(c);  
    }
```

```
    public void mul()  
    {  
        int a=10;  
        int b=20;  
        int c=a*b;  
        System.out.println(c);  
    }
```

```
s.o.println("Enter the second number ");  
b = scan.nextInt();  
  
int c = a+b;  
  
s.o.println(c);  
}  
  
public void sub()  
{  
    int a;  
    int b;  
  
    Scanner scan = new Scanner (System.in);  
    int a=scan  
  
    s.o.println ("Enter 1st number ");  
    int a = scan.nextInt();  
  
    s.o.println ("Enter 2nd number ");  
    int b = scan.nextInt();  
  
    int c = a - b;  
  
    s.o.println (c);  
}  
}  
  
class Calculator3 implements MyCalculator  
{  
    public void add()  
    {  
        int a;  
        int b;
```

Scanner scan = new Scanner (System.in);

s.o.println ("Enter first number");

a = scan.nextInt();

s.o.println ("Enter second number");

b = scan.nextInt();

if (a == 0 || b == 0)

{

s.o.println ("Provide valid input");

}

else

{

int c = a + b;

s.o.println (c);

}

}

public void sub()

{

int a;

(a) Addition sum = a + b

(b) Subtraction sum = a - b

int b;

(c) Multiplication sum = a * b

(d) Division sum = a / b

(e) Modulus sum = a % b

Scanner scan = new Scanner (System.in);

s.o.println ("Enter first number");

a = scan.nextInt();

s.o.println ("Enter second number");

b = scan.nextInt();

if (a == 0 || b == 0)

{

s.o.println ("Provide valid input");

}

else

{

int c = a - b;

s.o.println (c);

}

}

}

}

class Launch Math

{

public void Permit (MyCalculator ref) {
for more so

{

ref.add();

ref.Sub();

}

class Launch

{

public static void main (String args[]) {

{

Calculator1 c1 = new Calculator1();

Calculator2 c2 = new Calculator2();

Calculator3 c3 = new Calculator3();

Math m = new Math();

MyCalculator ref;

m.Permit (c1);

m.Permit (c2);

m.Permit (c3);

}

Rule - 5 :-

Object of an interface can't be created. (or) In other words An Interface can't be instantiated.

Pgm
MyCalculator

```

<-
public void add();
public void sub();

```

Interface MyCalculator

```

{
    public void add();
    public void sub();
}

```

class Launch

```

{
    public static void main(String args[])
    {
        ...
    }
}
```

Error → MyCalculator obj = new MyCalculator();

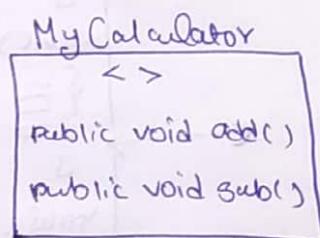
Rule - 6 :- Even if creation of object is not possible, reference of an interface can be created. In order to achieve loose coupling.

Pgm

```

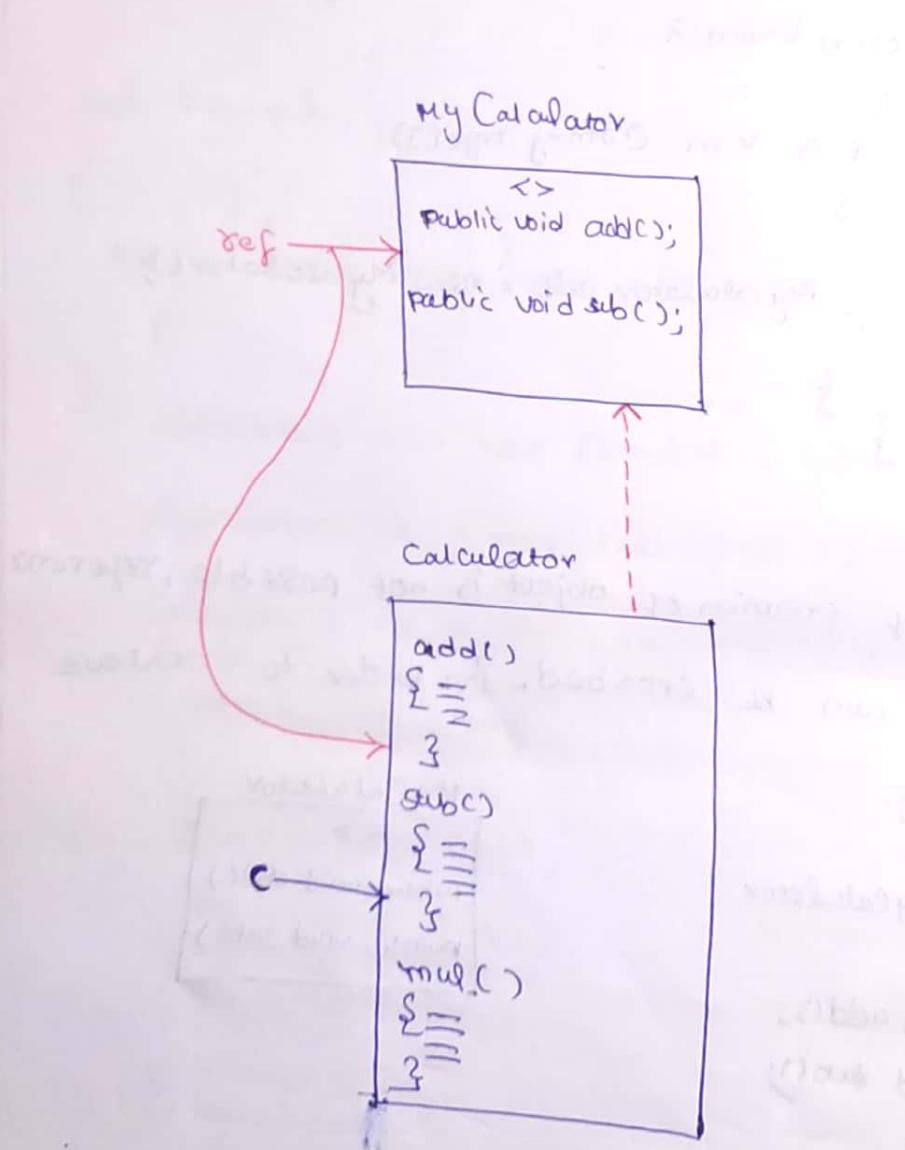
interface MyCalculator
{
    public void add();
    public void sub();
}

class Launch
{
    public static void main(String args[])
    {
        MyCalculator ref;
    }
}
```



Rule - 7

Using Interface type reference the overridden methods of the implementing class can be accessed directly. In order to access the specialized methods and properties of implementing class downcasting has to be performed.



Program

interface MyCalculator

{

 public void add();

 public void sub();

}

class Calculator implements MyCalculator

{

 public void add()

{

 int a = 10;

 int b = 20;

 int c = a + b;

 System.out.println(c);

}

 public void sub()

{

 int a = 10;

 int b = 20;

 int c = a - b;

 System.out.println(c);

}

 public void mul()

{

 int a = 10;

 int b = 20;

 int c = a * b;

 System.out.println(c);

}

class Launch

{

 static public void main C String args [])

{

 MyCalculator c1;

 Calculator c2 = new Calculator();

 c1 = c2;

 c1.add();

 c1.sub();

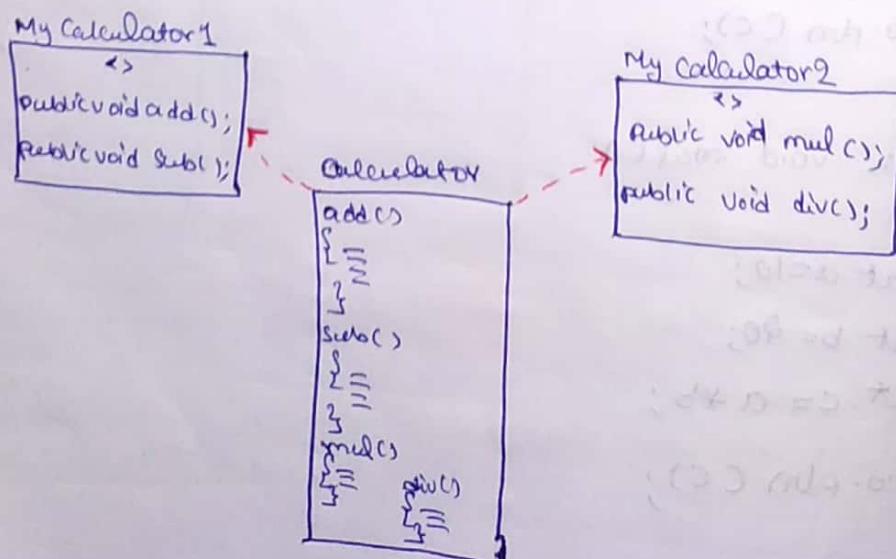
 ((Calculator) (c1)).mul();

}

}

Rule-8

A single class can implement multiple interfaces through interfaces in a way multiple inheritance is permitted indirectly.



interface MyCalculator1

```
{  
    public void add();  
    public void sub();  
}
```

interface MyCalculator2

```
{  
    public void mul();  
    public void div();  
}
```

class Calculator implements MyCalculator1, MyCalculator2

```
{  
    public void add()  
    {  
        int a = 10;  
        int b = 20;  
        int c = a + b;  
        System.out.println(c);  
    }  
}
```

```
public void sub()  
{  
    int a = 10;  
    int b = 20;  
    int c = a - b;  
    System.out.println(c);  
}
```

```
public void mul()  
{  
    int a = 10;  
    int b = 20;  
    int c = a * b;  
    System.out.println(c);  
}
```

```
public void div()
```

```
{ int a = 20;  
    int b = 10;  
    int c = a/b;  
    System.out.println(c);  
}
```

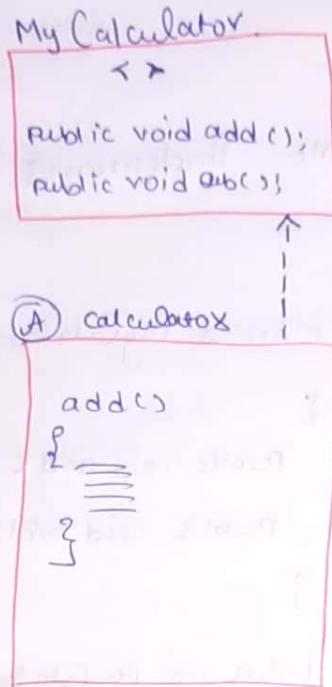
Class Launch

```
{  
    public static void main(String[] args)  
    {  
        Calculator c = new Calculator();  
        c.add();  
        c.sub();  
        c.mul();  
        c.div();  
    }  
}
```

Output :-
30
-10
200
2

Rule-9

Partial implementation of an interface is possible. But however if a class is partially implementing an interface it had to be declared as abstract.



Program :-

interface MyCalculator

{

 public void add();

 public void sub();

}

abstract class Calculator implements MyCalculator

{

 public void add()

{

 int a=10;

 int b=20;

 int c=a+b;

 System.out.println(c);

}

}

class Launch

{

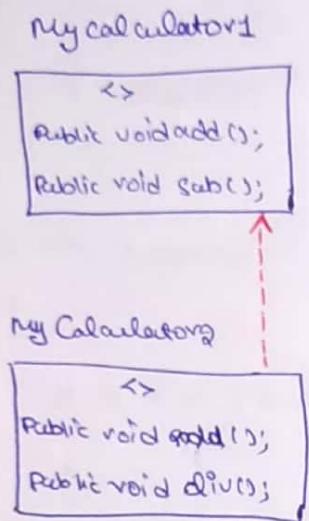
 public static void main(String args[])

{

}

Rule - 10

One Interface can't implements another Interface.



Err

Interface MyCalculator1

{

 public void add();

 public void sub();

}

interface MyCalculator implements

MyCalculator1 // Error

{

 public void mul();

 public void div();

}

class Launch

{

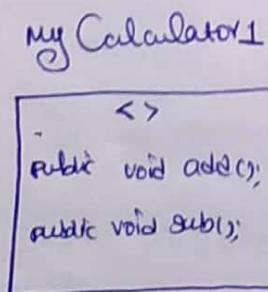
 psvm (String args[])

{

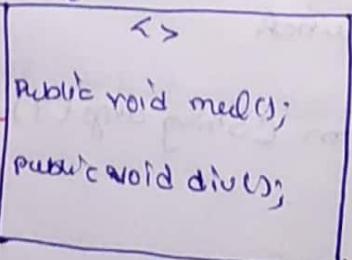
{

Rule - 11

One Interface can extends an another Interface



MyCalculator2



Interface MyCalculator1

```
{ public void add();  
    public void sub();  
}
```

Interface MyCalculator2 extends MyCalculator1

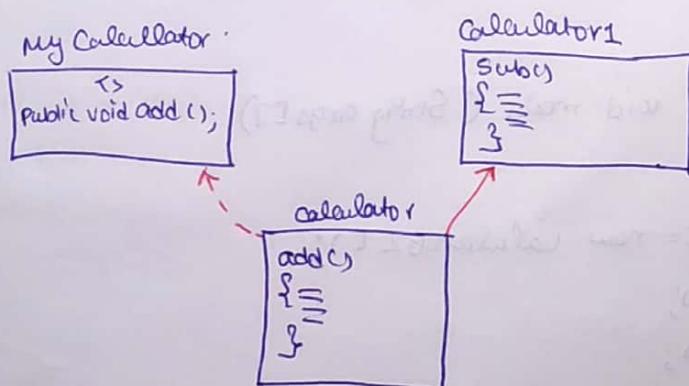
```
{ public void mul();  
    public void div();  
}
```

```
class Launch
```

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

Rule-12

A single class can both implement an interface and extend another class. But, however extends relationship should be established first later implements relationship should be established. In other words a class has to be extends first another class later implements the interface.



Program

Calculator.java

Interface MyCalculator

```
{
    public void add();
}

class Calculator1 {
{
    public void sub() {
        {
            int a=10;
            int b=20;
            int c=a+b;
            System.out.println(c);
        }
    }
}
```

class Calculator2 extends Calculator1 implements MyCalculator

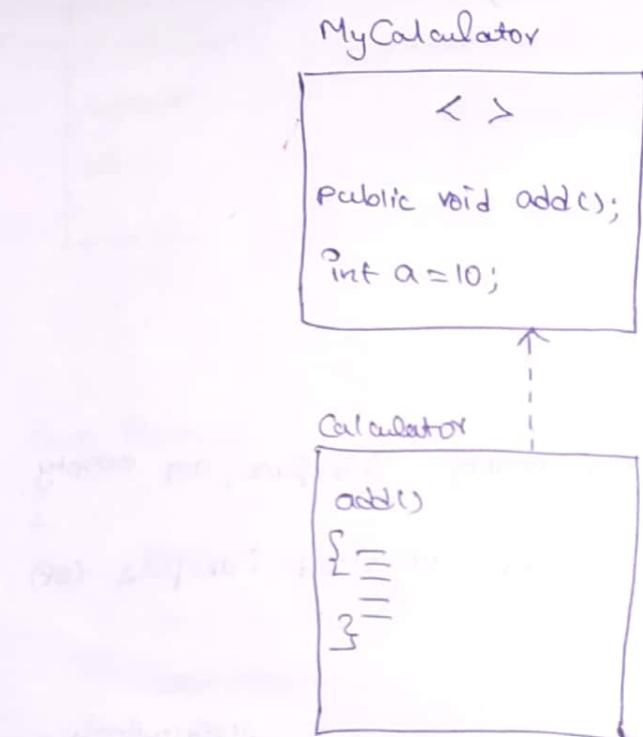
```
{
    public void add() {
        {
            int a=10;
            int b=20;
            int c=a+b;
            System.out.println(c);
        }
    }
}
```

Class Launch

```
{
    static public void main (String args[]) {
        {
            Calculator2 c = new Calculator2 ();
            c.add();
            c.sub();
        }
    }
}
```

Rule - 13

Inside an interface variables can be declared, however all the variables declared inside a interface are automatically "final".



Pg 8 M

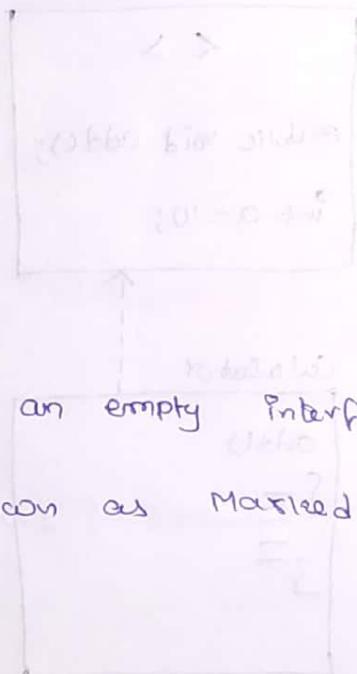
```
interface MyCalculator
{
    int a = 10;
    public void add();
}

class Calculator implements MyCalculator
{
    public void add()
    {
        System.out.println(a);
    }
}
```

```

class Launch
{
    public static void main (String [] args)
    {
        Calculator c = new Calculator ();
        c.add ();
    }
}

```



Rule - 14

We can create an empty Interface, an empty Interface is also known as Marked interface or Tagged interface.

Program

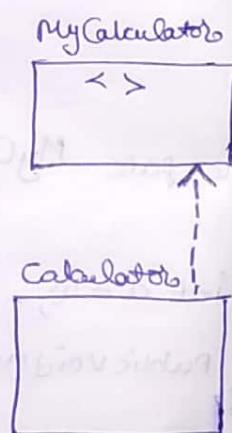
```

interface MyCalculator
{
}

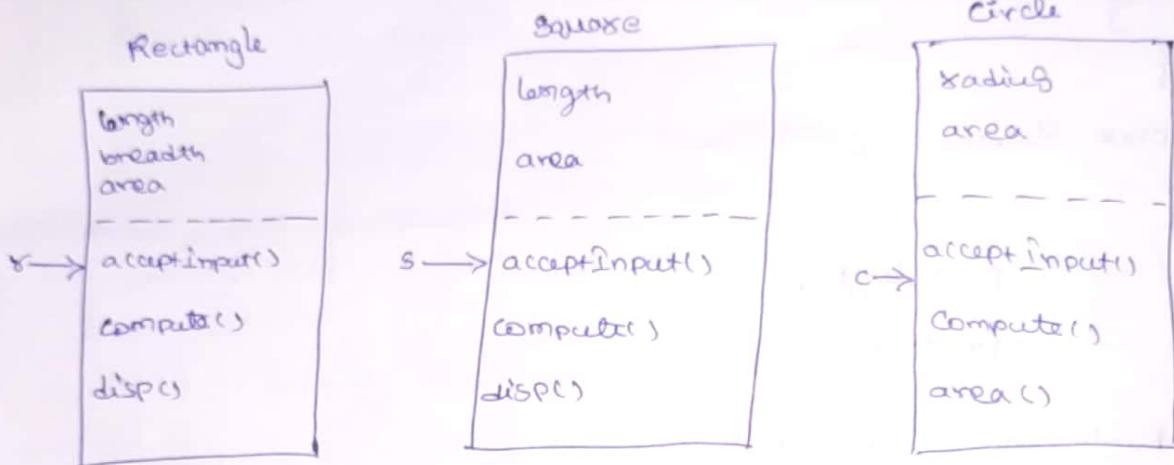
class Calculator implements MyCalculator
{
}

class Launch
{
    public static void main (String ... args)
}

```



Non-Object Oriented Approach



```
class Rectangle
{
    float length;
    float breadth;
    float area;
    public void acceptInput()
    {

```

```
        Scanner scan = new Scanner(System.in);
    }
```

```
    System.out.print("Enter the length");

```

```
    length = scan.nextFloat();

```

```
    System.out.print("Enter the breadth");

```

```
    breadth = scan.nextFloat();

```

```
}
```

```
    public void compute()
    {

```

```
        area = length * breadth;
    }
}
```

```

    public void disp()
    {
        System.out.println("area is :" + area);
    }
}

class Square
{
    float length;
    float area;

    public void accept_Input()
    {
        Scanner Scan = new Scanner (System.in);
        System.out.println("Enter the length");
        length = Scan.nextFloat();
    }

    public void compute()
    {
        area = length * length;
    }

    public void disp()
    {
        System.out.println("area is :" + area);
    }
}

class Circle
{
    float radius;
    float area;

    public void accept_Input()

```

```

}

Scanner scan = new Scanner(System.in);

System.out.println("Enter the radius");
radius = scan.nextDouble();

}

public void compute()
{
    area = 3.14 * radius * radius;
}

public void disp()
{
    System.out.println("Area is :" + area);
}

```

```

class Launcher
{
    public static void main(String args[])
    {
        Rectangle r = new Rectangle();
        Square s = new Square();
        Circle c = new Circle();

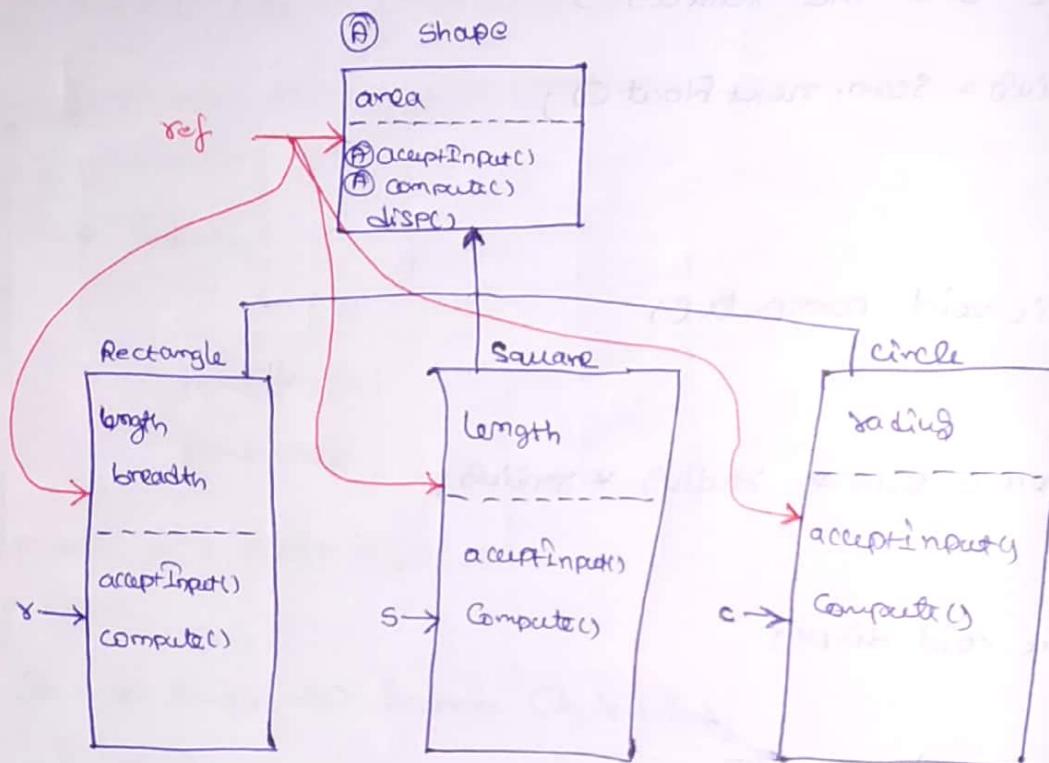
        r.acceptInput();
        r.compute();
        r.disp();

        s.acceptInput();
        s.compute();
        s.disp();

        c.acceptInput();
        c.compute();
        c.disp();
    }
}

```

Object Oriented Program approach



(A) Class Shape

```
{  
    float area;  
    public void  
    abstract acceptInput();  
    public void  
    abstract compute compute();  
}  
disp()
```

```
{  
    System.out.println("area is :" + area);  
}
```

```
}
```

```
class Rectangle extends Shape
```

```
{
```

```
    private float length;
```

```
    private float breadth;
```

```
public void acceptInput()
{
    Scanner Scan = new Scanner(System.in);
    S.o.p("Enter the length");
    length = Scan.nextFloat();
    S.o.p("Enter the breadth");
    breadth = Scan.nextFloat();
}

public void compute()
{
    area = length * breadth;
}

class Square extends Shape
{
    private float length;
    public void acceptInput()
    {
        Scanner Scan = new Scanner (System.in);
        S.o.p("Enter the length");
        length = Scan.nextFloat();
    }
    public void compute()
    {
        area = length * length;
    }
}
```

class Circle extends Shape

{
 private float radius;

 public void acceptInput(){

{
 Scanner scan = new Scanner (System.in);

 System.out.print("Enter the radius");

 radius = scan.nextFloat();

}

 public void compute()

{

 area = 3.14 * radius * radius;

}

class Geometry

{

 public void permit(Shape ref)

{
 ref.acceptInput();

 ref.compute();

 ref.display();

}

class Launch

{

 public static void main (String args[])

{

 Rectangle r = new Rectangle();

 Square s = new Square();

 Circle c = new Circle();

Geometry g = new Geometry

g.permit(k);

g.permit(s);

g.permit(c);

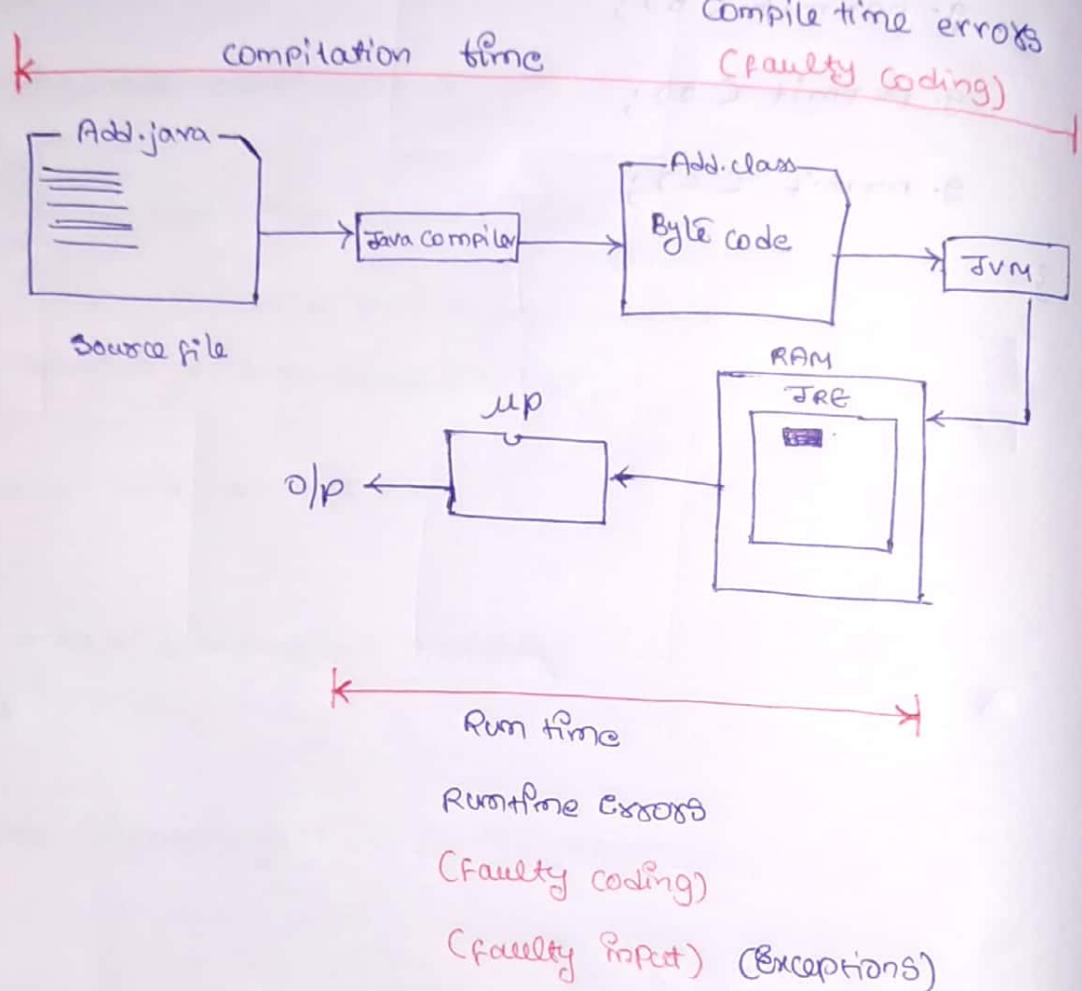


(newGeometry) == oldG

initial set more points crosses permit no problem
but when, remove what got to move from value
memory set to address of new object
and set to crossing set to the old one, result
between the two will be

the problem is always to be printed memory is
deallocate each time so have to understand

Exception Handling



- * Exceptions are runtime errors which occur due to the faulty input given by the user, exception results in abrupt termination of the program.
- * Hence, Exception affects the performance of the software if they are not handled.
- * Exception handling is a process of handling an exception in such a way that abrupt

Termination of the program is avoided and normal termination of the program is achieved.

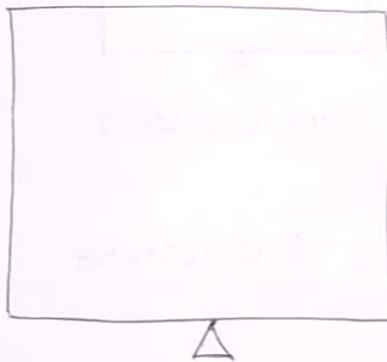
- * If the generated exception is handled successfully then normal termination occurs and the performance of the software/program is not affected.

1940's

1950's

1960's

1970's



App development

App Compilation

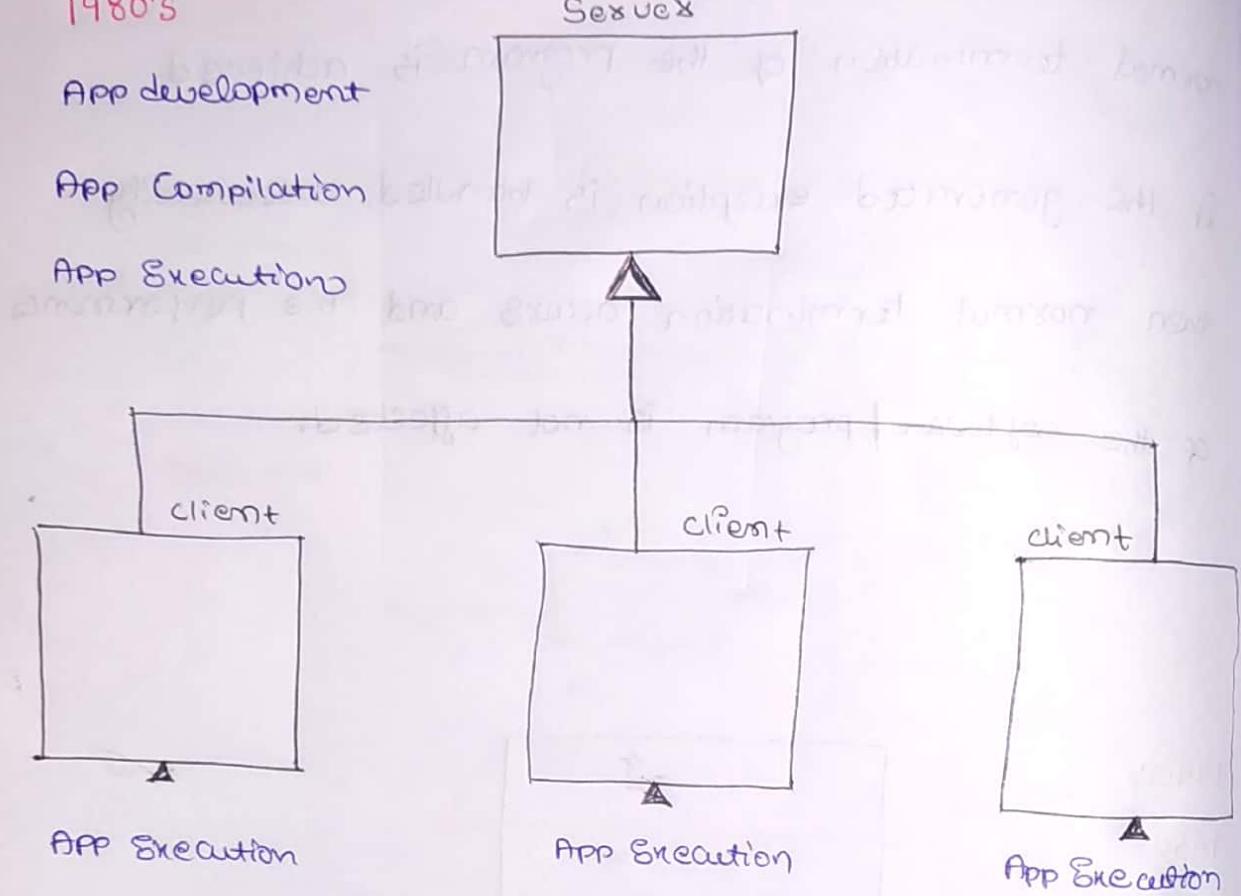
App Execution

Tier-1 Architecture

(OS)

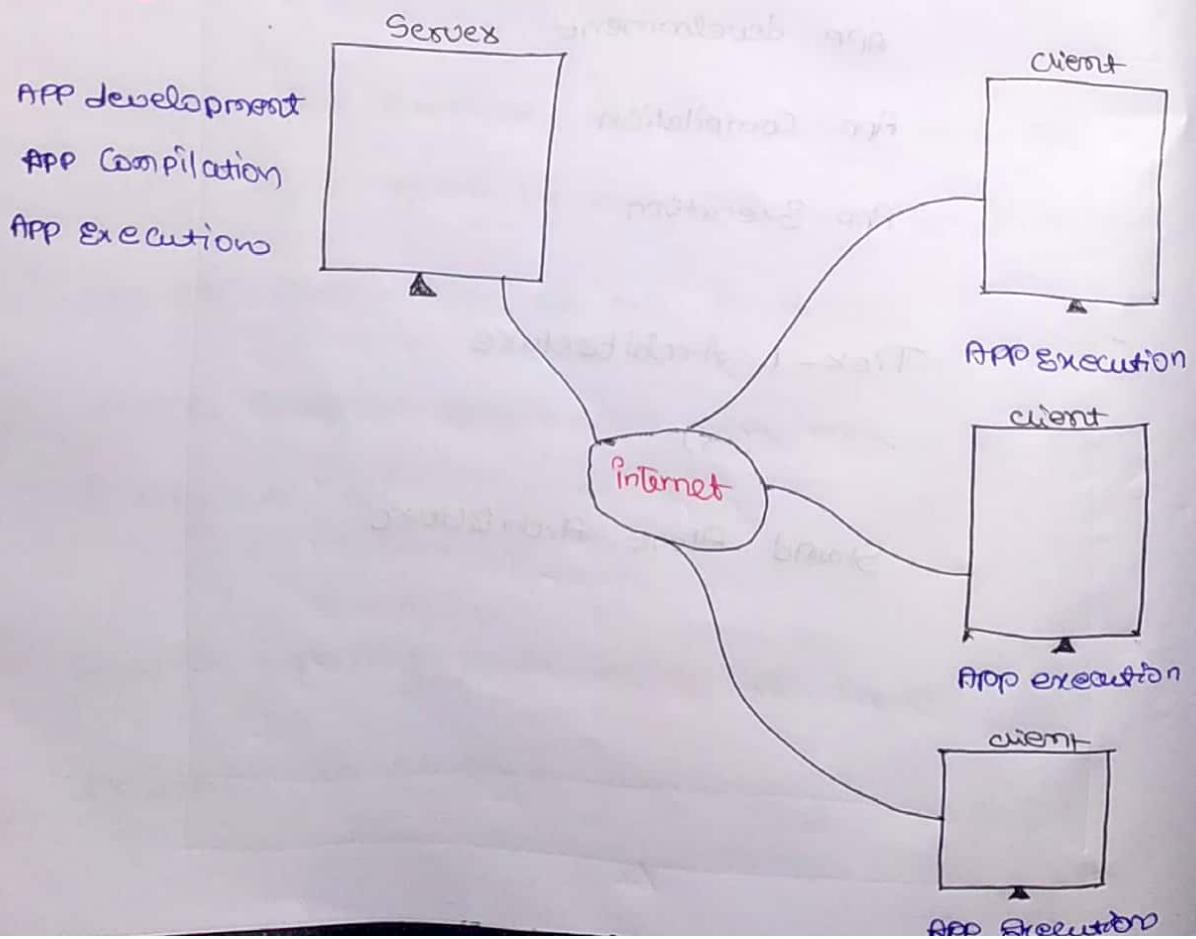
Stand Alone Architecture

1980's

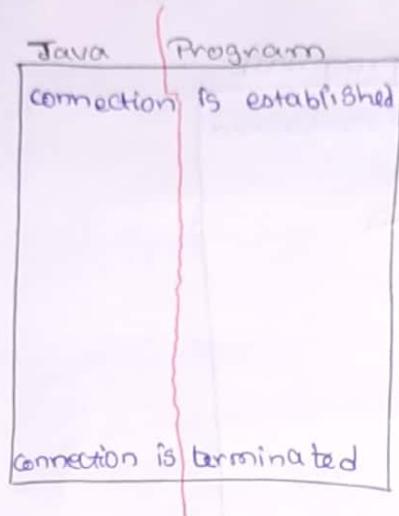


Tier-2 Architecture

1990's



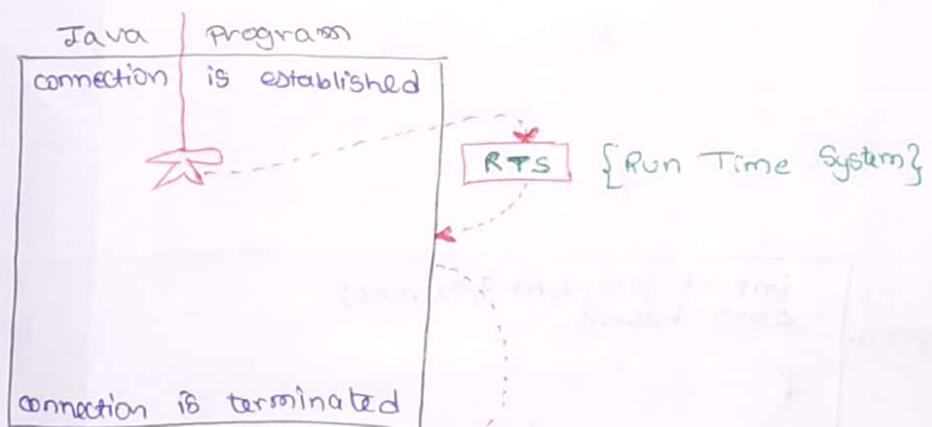
Case-I



Default Exception Handling

OS

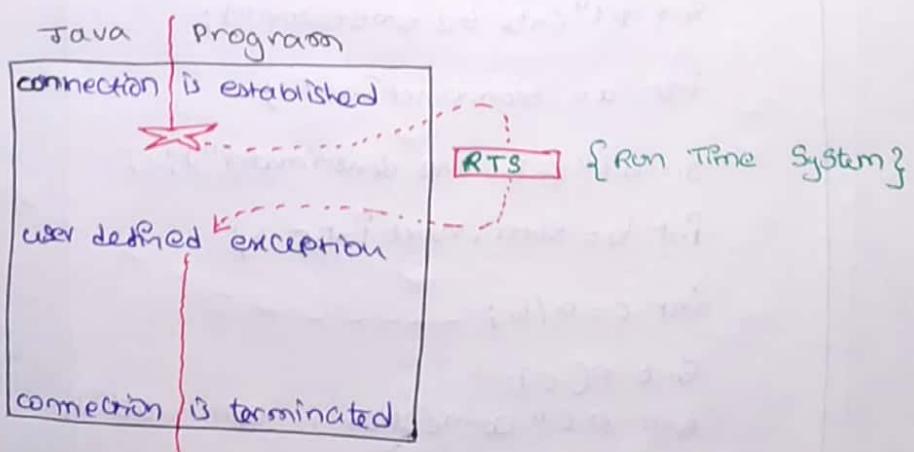
case-II



Default Exception Handling

OS

case-III



Default Exception
Handling

OS



O/S

System crash

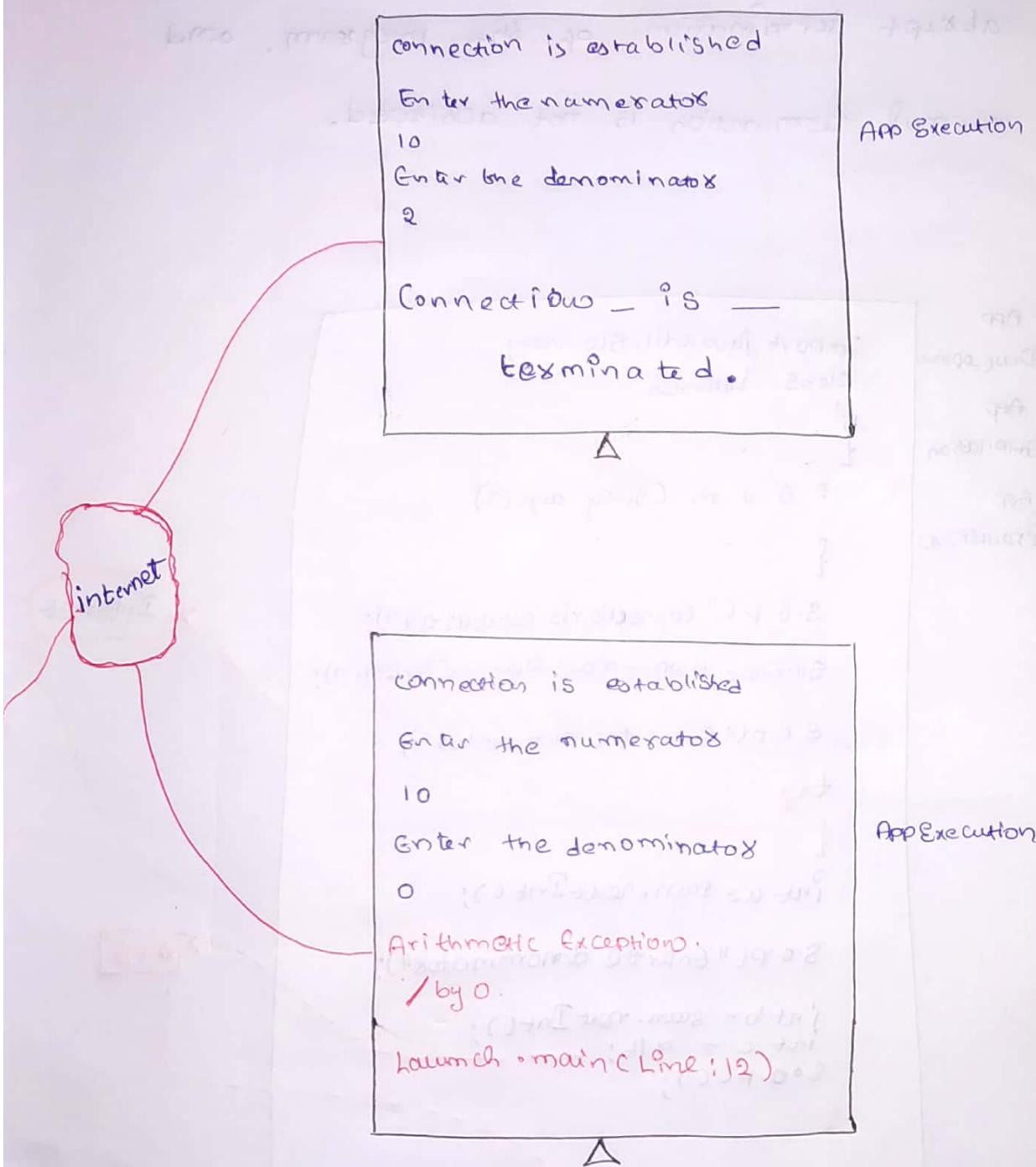
APP Development
APP Compilation
APP Execution.

```

import java.util.Scanner;
class launch
{
    public static void main (String args [])
    {
        System.out.println ("Connection is established");
        Scanner scan = new Scanner (System.in);
        System.out.print ("Enter the numerator ");
        int a = scan.nextInt();
        System.out.print ("Enter the denominator ");
        int b = scan.nextInt();
        int c = a/b;
        System.out.println (c);
        System.out.println ("Connection is terminated");
    }
}

```

RTS
DGH
OS



As shown in the above example, if a faulty input is given by the user exception is generated but however since the programmer has not defined any user defined exception handler (UDEH), the default exception handling (DEH) mechanism of java handles the exception but

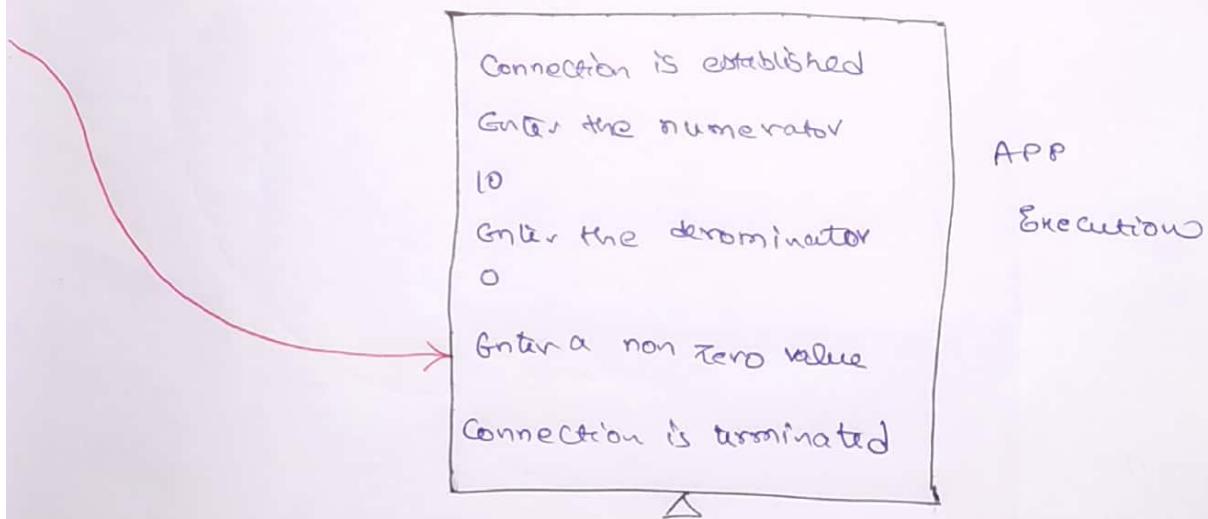
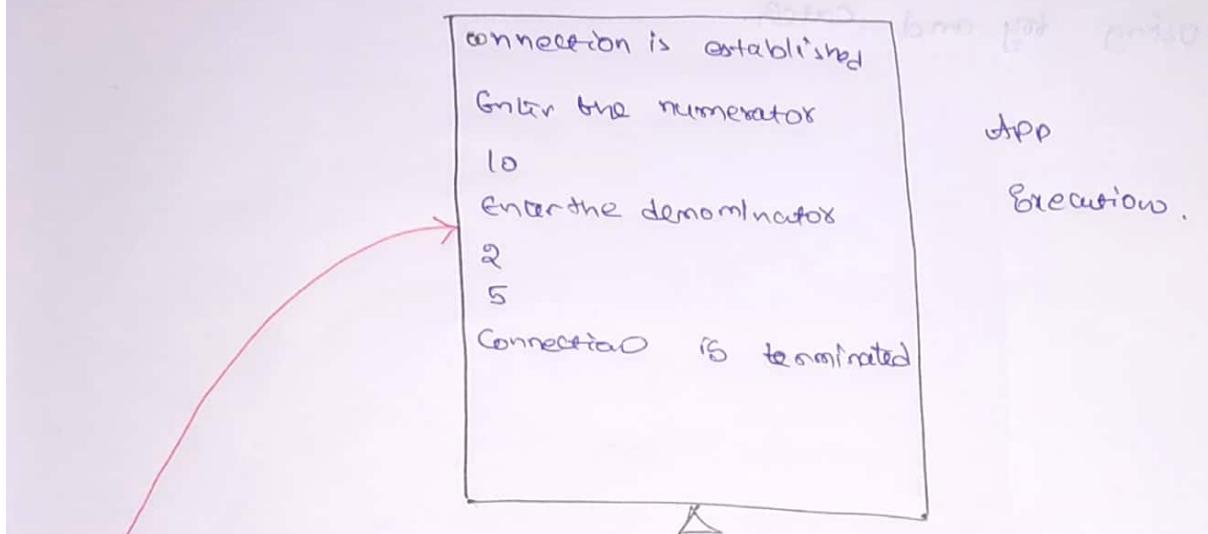
The ~~advantage~~ here is this process results in abrupt termination of the program. and normal termination is not achieved.

- APP Development
- APP Compilation
- APP Execution

```
import java.util.Scanner;  
class Launch  
{  
    public static void main (String args [])  
    {  
        System.out.println ("Connection is established");  
        Scanner scan = new Scanner (System.in);  
        System.out.println ("Enter the numerator");  
        try {  
            int a = scan.nextInt();  
            System.out.println ("Enter the denominator");  
            int b = scan.nextInt();  
            int c = a/b;  
            System.out.println (c);  
        }  
        catch (Exception e)  
        {  
            System.out.println ("Enter a non zero value");  
        }  
        System.out.println ("Connection is terminated");  
    }  
}
```

Interest

RTS



* In order to handle Exceptions in java we make use of few key words and they are try, catch, throw, throws finally. As shown in the above example an exception is generated due to faulty input but however the exception is not handled by D&H mechanism because the way of the programming

has defined user defined exception handler (UDEH)

using try and catch.

Disadvantage of single try-catch hierarchy

class Launch

{

 public static void main (String args [])

{

 System.out.println ("Connection is established");

 Scanner scan = new Scanner (System.in);

 System.out.println ("Enter the numerator");

 try

 {

 int a = scan.nextInt();

 System.out.println ("Enter the denominator");

 int b = scan.nextInt();

 int c = a/b;

 System.out.println (c);

 System.out.println ("Enter the size of an array");

 int size = scan.nextInt();

 int arr [] = new int [size];

 System.out.println ("Enter the position of arr where the

 element has to be stored");

 int pos = scan.nextInt();

```
s.o.p ("Enter the integer to be stored");
```

```
int element = scan.nextInt();
```

```
arr [pos] = element;
```

```
s.o.p (arr [pos]);
```

```
}
```

Catch (Exception e)

```
{
```

```
s.o.p ("Some Problem Occured");
```

```
}
```

```
s.o.p ("Connection is terminated");
```

```
}
```

```
}
```

As shown in the above program whatever might be the

reason for exception only a single error message is printed.

and this results in confusion in the minds of user because

the right reason behind the exception is not being -

conveyed. This is the disadvantage of having a

single catch block.

In order to resolve this problem we include multiple catch blocks which are capable of handling

specific exceptions, once specific exceptions are

handled by specific catch blocks a right error message can be printed. This is the advantage of multiple hierarchy.

Note :-

1. Multiple catch block hierarchy must have a generic catch block along with the specific catch blocks. In order to handle unanticipated exceptions are such exceptions which the programmer has never thought about.
2. A generic catch block has to be included at the bottom of the catch block hierarchy so that the specific catch blocks remains functional.
3. Generic catch block is such a catch block which is capable of handling all exceptions.
Specific catch block is such a catch block which can handle only specific exceptions.

Program

class launch

{

 public static void main (String args[]){

{

 System.out.println ("connection is established");

 Scanner scan = new Scanner (System.in);

 System.out.println ("Enter the Numerator");

 try

{

 int a = scan.nextInt();

 System.out.println ("Enter the Denominator");

 int b = scan.nextInt();

 int c = a/b;

 System.out.println (c);

 System.out.println ("Enter the size of an array");

 int size = scan.nextInt();

 int arr[] = new int [size];

 System.out.println ("Enter the position in which element to be stored");

 int pos = scan.nextInt();

 System.out.println ("Enter the integer to be stored");

 int elem = scan.nextInt();

 arr[pos] = elem;

 }

 catch (ArithmaticException ae)

{

 System.out.println ("Enter a non zero value");

}

catch (NegativeArraySizeException e)

{

s.o.p ("Enter a positive value");

}

catch (ArrayIndexOutOfBoundsException e)

{

s.o.p ("Enter the value within the size");

}

catch (Exception e)

{

s.o.p ("Some problem occurred");

}

}

Behaviours of Run-Time System

Example

class Demo

{

public void alpha()

{

s.o.p ("connection is established");

Scanner scan = new Scanner (System.in);

s.o.p ("Enter a numerator");

int a = scan.nextInt();

s.o.p ("Enter a denominator");

int b = scan.nextInt();

int c = a/b;

s.o.p(c);

{ s.o.p ("Connection is terminated"); }

class launch

{

PSVM (String args [])

{

SIO_PCL("Connection1 is established");

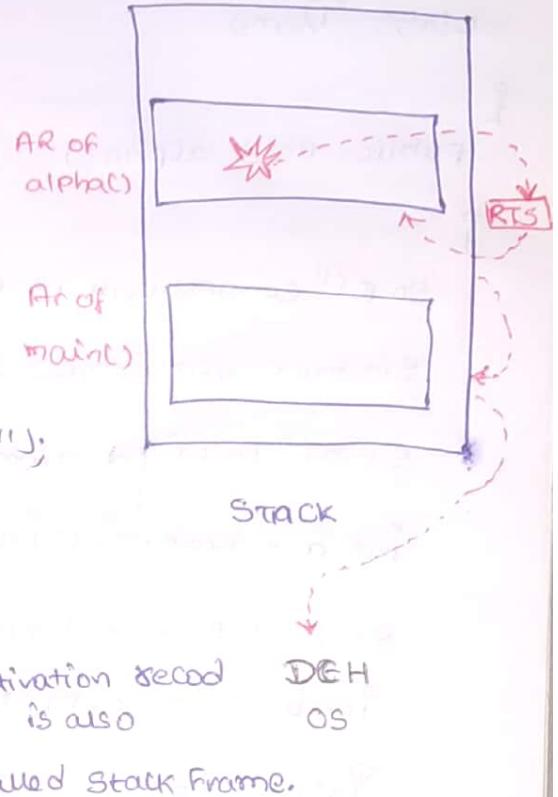
Demod = new Demod();

di_alpha();

SIO_PCL("Connection 1 is terminated");

{

}



→ As shown in the above program if an exception gets generated in a method and if the same method doesn't handle the exception then the control is not given directly to the DEH mechanism before the control is given to DEH theollar of the method is checked only and only if both the methods don't have UDEH then the control is given to DEH.

Example-8

```
class Demo
{
    public void alphac()
    {
        System.out.println("Connection 1 is established");
        Scanner scan = new Scanner(System.in);
        System.out.println("Enter a numerator");
        int a = scan.nextInt();
        System.out.println("Enter a denominator");
        int b = scan.nextInt();
        int c = a/b;
        System.out.println(c);
        System.out.println("Connection 1 is terminated");
    }
}

class Demo2
{
    public void betac()
    {
        System.out.println("Connection 2 is established");
        Demo d = new Demo();
        d.alphac();
        System.out.println("Connection 2 is terminated");
    }
}

class Demo3
{
    public void gammac()
    {
        System.out.println("Connection 3 is established");
        Demo2 d2 = new Demo2();
        d2.betac();
    }
}
```

```
sio.p("connection 1 is terminated");
```

```
}
```

```
class launch
```

```
{
```

```
p s v m CString args[2]
```

```
{
```

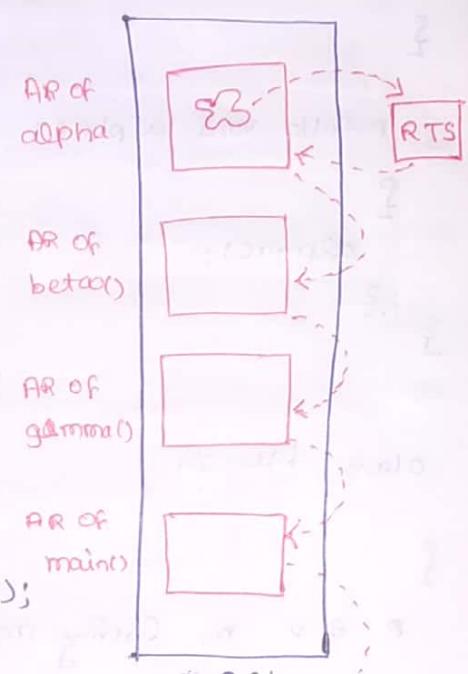
```
sio.p("connection 1 is established");
```

```
Demo3 d3 = new Demo3();
```

```
d3.gammain();
```

```
sio.p("connection 1 is terminated");
```

```
}
```



DEH
OS

As shown in the above program the entire exception stack hierarchy is traced and only if the UDEH is not found in the entire stack hierarchy then the exception is handled by DEH.

Programs

class Demo

{

 public void alphac()

{

 alphac();

}

class Launch

{

 public static void main (String args [])

{

 Demo d = new Demo();

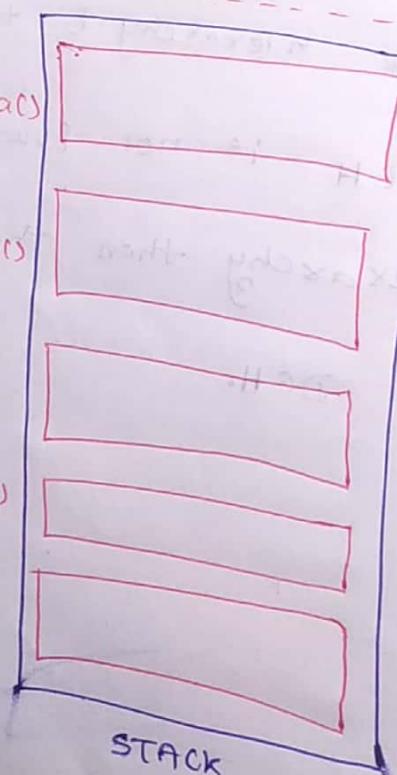
}

 d.alphac();

}

AR of
alphac()

AR of
main()



Exception vs Error

Exception

- It can be handled using try - catch.
- All exceptions are run-time errors and they occur only during run-time.

Eg:- Arithmetic Exception
Array Index Out Of Bounds

Error

- It can't be handled using try - catch.
- Errors can occur during both compile time and run-time.

Eg:- Out Of Memory Error

Stack Overflow Error

Generic catch block vs Specific catch block

Generic catch block

- It has exception type reference.
- Can handle all exceptions.
- It is present at the bottom of multiple catch block hierarchy.

Specific catch block

- It doesn't have exception type reference. It has the reference of any child class of exception class.
- Can handle only specific exceptions.
- It is present above the generic catch block.

User Defined Exception Handler Vs Default Exception

Handling Mechanism for

UDH

- It results in normal termination of a program.
- Programmers friendly specific error message can't be printed.
- UDH - User Defined

Exceptions Handling.

DGH Mechanism

- It results in abrupt termination of a program.
- Specific error message is printed.
- DGH Mechanism - Default Exception Handling Mechanism.

UDEH

- It results in normal termination of a program.
- Programmer friendly specific error message can't be printed.
- UDEH - User Defined Exception Handler.

DGH Mechanism

- It results in abrupt termination of a program.
- Specific error message is printed.
- DGH Mechanism - Default Exception Handling Mechanism.

Different ways of Handling Exceptions :-

1. Handling an Exception.
2. Rethrowing an Exception.
3. Ducking an Exception.

Handling an Exception :-

If the exception is handled in the same method where it is generated then this technique of handling an exception is called "Handling an Exception".

Program

```
class Demo
```

```
{
```

```
    public void alpha()
```

```
{
```

```
        System.out.println("Connection is established");
```

```
        Scanner scan = new Scanner(System.in);
```

```
        System.out.println("Enter the numerator");
```

```
    try
```

```
{
```

```
        int a = scan.nextInt();
```

```
        System.out.println("Enter the denominator");
```

```
        int b = scan.nextInt();
```

```
        int c = a/b;
```

```
        System.out.println(c);
```

```
}
```

Arithmatic

```
    catch (Exception e)
```

```
{
```

```
        System.out.println("Enter a non zero value");
```

```
}
```

```
    catch (Exception e)
```

```
{
```

```
        System.out.println("Some problem occurred");
```

```
}
```

```
    System.out.println("Connection 2 is terminated");
```

```
}
```

```
class Launch
```

```
{
```

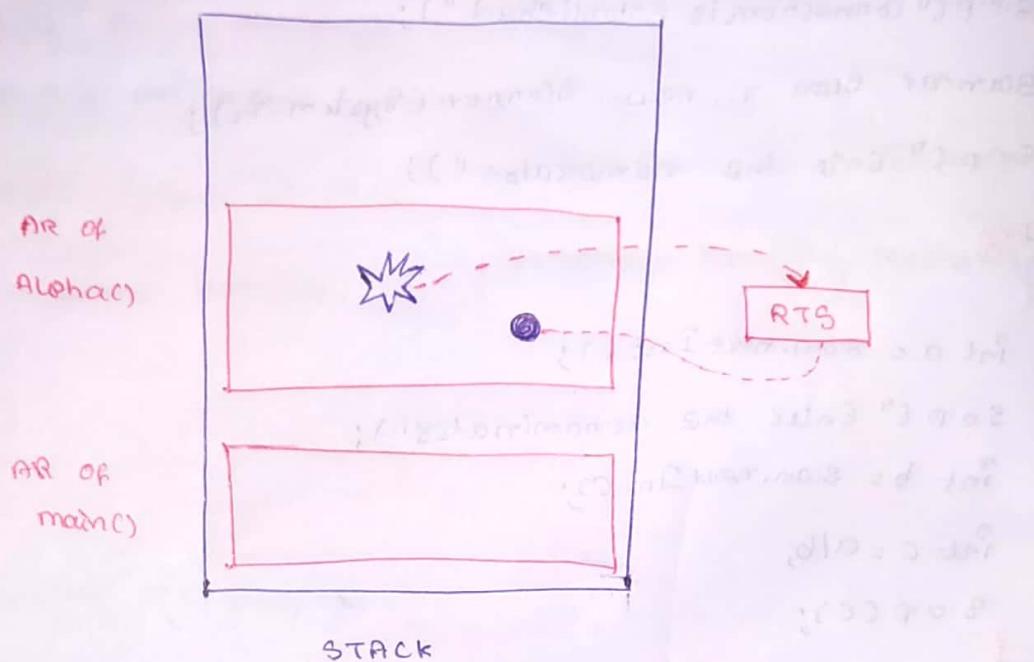
```
    public static void main (String args[])
```

```
{
```

```
    System.out.println("Connection 1 is established");
```

```
    Demo d = new Dem
```

```
d.alpha();  
S.O.P ("Connection is terminated");  
}  
}
```



Re throwing an Exception :-

In rethrowing an exception technique the exception i.e., handled in a method is forcefully rethrown down the stack and the called method again handles the already handled exception. Throw keyword is used to forcefully throw the already handled exceptions.

```
class Demo
```

```
{
```

```
    public void alphac() throws Exception
```

```
{
```

```
        System.out.println("Connection is established");
```

```
        Scanner scan = new Scanner(System.in);
```

```
        System.out.println("Enter the numerator");
```

```
        int a = scan.nextInt();
```

```
{
```

```
        int b = scan.nextInt();
```

```
        System.out.println("Enter the denominator");
```

```
        int c = a/b;
```

```
        System.out.println(c);
```

```
}
```

```
    catch (Exception e)
```

```
{
```

```
        System.out.println("Exceptions handled in alphac());
```

```
} throws;
```

```
finally
```

```
{
```

```
    System.out.println("Connection is terminated");
```

```
}
```

```
}
```

```
class Launch
```

```
{
```

```
    public static void main (String args [])
```

```
{
```

```
    System.out.println("Connection 1 is established");
```

```
    Demo d = new Demo();
```

try

{

d. alpha();

}

Catch (Exception e)

{

so if C "exception is handled in main()",

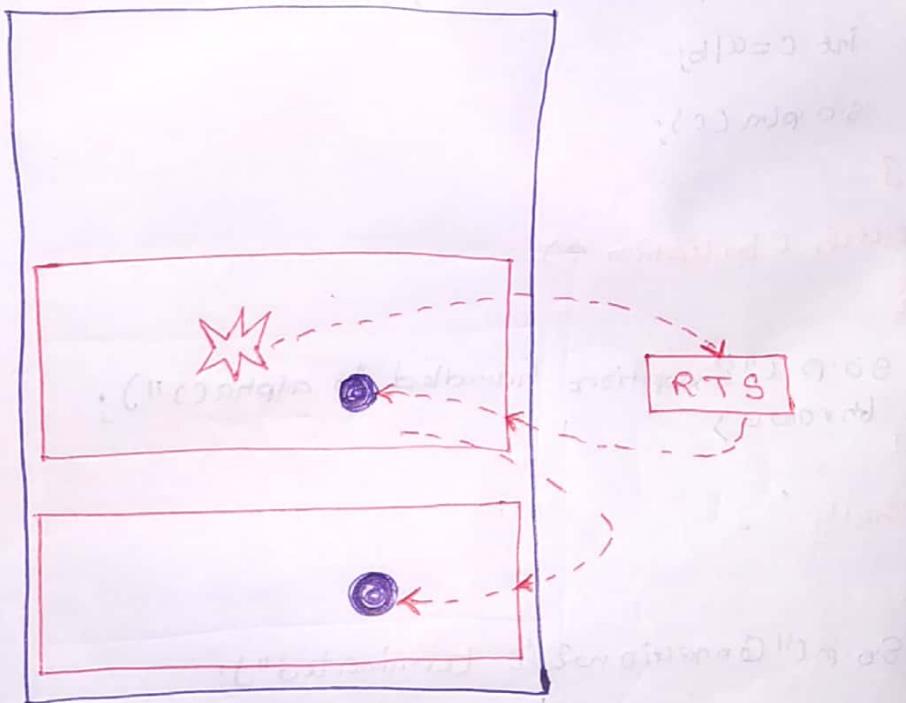
}

}

}

AR of
alpha()

AR of
main()



D G H

OS

During an Exception, In Handling an Exception

the exception is not handled in the method where it gets generated but however the exception is handled by the caller of the method.

Program

```
class Demo
{
    public void alphac() throws Exception
    {
        System.out.println("Connection is established");
        Scanner Scan = new Scanner(System.in);
        System.out.println("Enter the numerator");
        try
        {
            int a = Scan.nextInt();
            System.out.println("Enter the denominator");
            int b = Scan.nextInt();
            System.out.println(a/b);
            System.out.println();
        }
        finally
        {
            System.out.println("Connection is terminated");
        }
    }
}
```

Class Launch

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

```
s.o.p ("connection1 is established");
```

```
    demo = new Demo();
```

boy

{

```
    d.alphac();
```

}

```
catch (Exception e)
```

{

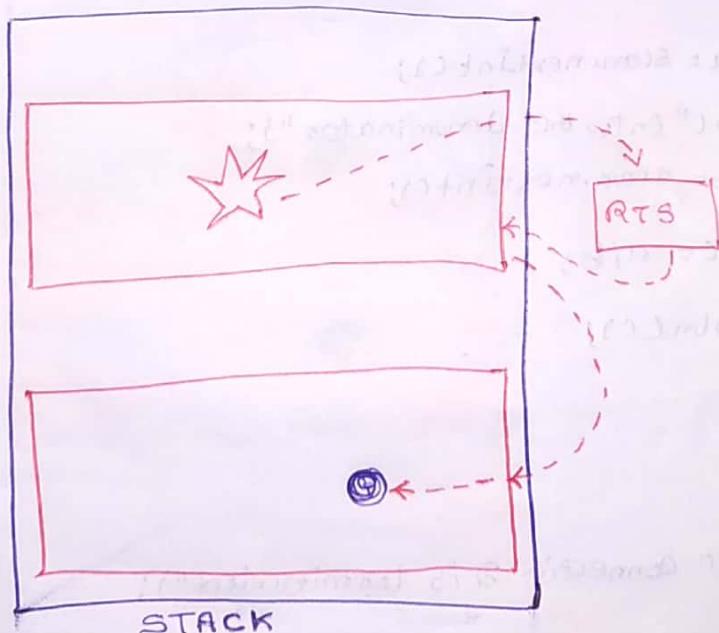
```
s.o.p ("Exception handled @ main()");
```

}

```
s.o.p ("connection1 is terminated");
```

}

AR of
alphac()



DGH

OS

Legal and illegal combinations of try, catch & finally.

Legal

```
* try
  {
    ==
  }
  catch ( )
  {
    ==
  }
```

* try
{ }
}
catch()
{ }
}
finally
{ }
}

* try
{ }
Finally
{ }

* try
{ == }
} catch(c)
{ == } try
{ == }
} catch(c)
{ == }

- * boy
 - { ==
 - ==
 - ==
- boy
 - { ==
 - ==
 - ==
- catch
 - { ==
 - ==
 - ==
- Finally
 - { ==
 - ==
 - ==

* try
{ =
} =
} =
catch()
{ =
} =
} =
finally
{ =

Finally
L =
3

```
Catch( )  
{  
    //  
    //
```

finally
{ =
}

Illegal

* try

```
{  
= = =  
}  
=
```

* catch()

```
{  
= = =  
}  
=
```

* finally

```
{  
= = =  
}  
=
```

* catch()

```
{  
= = =  
}  
=
```

* try

```
{  
= = =  
}  
=
```

finally

```
{  
= = =  
}  
=
```

finally

```
{  
= = =  
}  
=
```

catch()

```
{  
= = =  
}  
=
```

Note :-

→ Inorder to handle an exception in java. we have five keywords. They are try, catch, throw, throws, finally.

→ try block is used to include such risky codes which might generate exception.

→ catch block is used to handle the exception that's get generated in try block.

→ Throw keyword is used to forcefully rethrow the handled exception.

→ Throws keyword is used to duck a checked exception.

→ If a part of code has to be executed without fail irrespective of whether the exception is getting generated or not then that part of the code should be included within finally block.

Custom Exception:

```
class ATM  
{  
    private int acc-no = 3333;  
    private int pwd = 4444;  
    private int acc;  
    private int Pass;  
  
    public void acceptInput()  
{  
        Scanner scan = new Scanner(System.in);  
        System.out.println("Enter the account number");  
        acc = scan.nextInt();  
        System.out.println("Enter the password");  
        Pass = scan.nextInt();  
    }  
  
    public void verify()  
{  
        if (acc-no == acc & pwd == Pass)  
        {  
            System.out.println("Collect your money");  
        }  
        else  
        {  
            System.out.println("Invalid credentials!! Try again");  
        }  
    }  
}
```

```

class Bank
{
    public void initiate()
    {
        ATM atm = new ATM();
        atm.acceptInput();
        atm.verify();
    }
}

class Launch
{
    public static void main(String args[])
    {
        Bank b = new Bank();
        b.initiate();
    }
}

```

Note

- The inbuilt Exception Handling facility of java fails to generate an exception in all cases. The inbuilt Exception Handling facility of java generates an exception for only standard faulty inputs.
- Whenever the inbuilt facility fails to generate the exception the programme has to generate the exception in order to meet the needs of project.

Such user defined exception created by programmer

to meet the needs of project is only called as
"custom exception."

→ As shown in the below program the programmer has created an exception object when required since exception has not generated automatically by Java.

Java's inbuilt facility and the "Invalid Input Exception"

that is generated by the programmer to satisfy the needs of the below mentioned program and

is only the custom exception i.e., created by programmer

program :-

Exception

getMessage()

{

 }

PrintStackTrace()

{

 }

Invalid Input Exception

public String getMessage()

{
 return "Invalid credentials";
}

"try again".

class InvalidInputException extends Exception

```
{  
    public String getMessage()  
    {  
        return "Invalid credentials !! Try again !!";  
    }  
}
```

class ATM

```
{  
    private int acc_no = 3333;
```

```
    private int pwd = 4444;
```

```
    private int acc;
```

```
    private int pass;
```

```
    public void accept Input()
```

```
{
```

```
    Scanner Scam = new Scanner (System.in);
```

```
    S.O.P ("Enter the account number");
```

```
    acc = Scam.nextInt();
```

```
    S.O.P ("Give the password");
```

```
    pass = Scam.nextInt();
```

```
}
```

```
    public void verify () throws Exception
```

```
{  
    if (acc_no == acc & & pwd == pass)
```

```
{
```

```
        S.O.P ("Collect your money");
```

```
}  
    else  
    {
```

InvalidInputException ie = new InvalidInputException.
 ie.getMessage();

```
        S.O.P (ie.getMessage());
```

```
}  
    }  
    throw ie;
```

Clothes Bank

{ periodic void initiate ()

$$\{ \text{ATM atom} = \text{new ATM C},$$

68

atm.acceptInout();
 atm.verify();

3

Catch (Exception e)

8

try

۸

atm.acceptInput();

atm.verify();

21

catch (Exception e)

۸

try

۸۲

soft adj. acceptative

2

attr. ready();

catch (exception g)

S

s.o.p ("card is blocked"));

```
S.o.p.ln System.exit(0);
```

2

3

?

8

class launch

{
 p sum (CString args[])

{
 Bank b = new Bank();

 b.initiate();

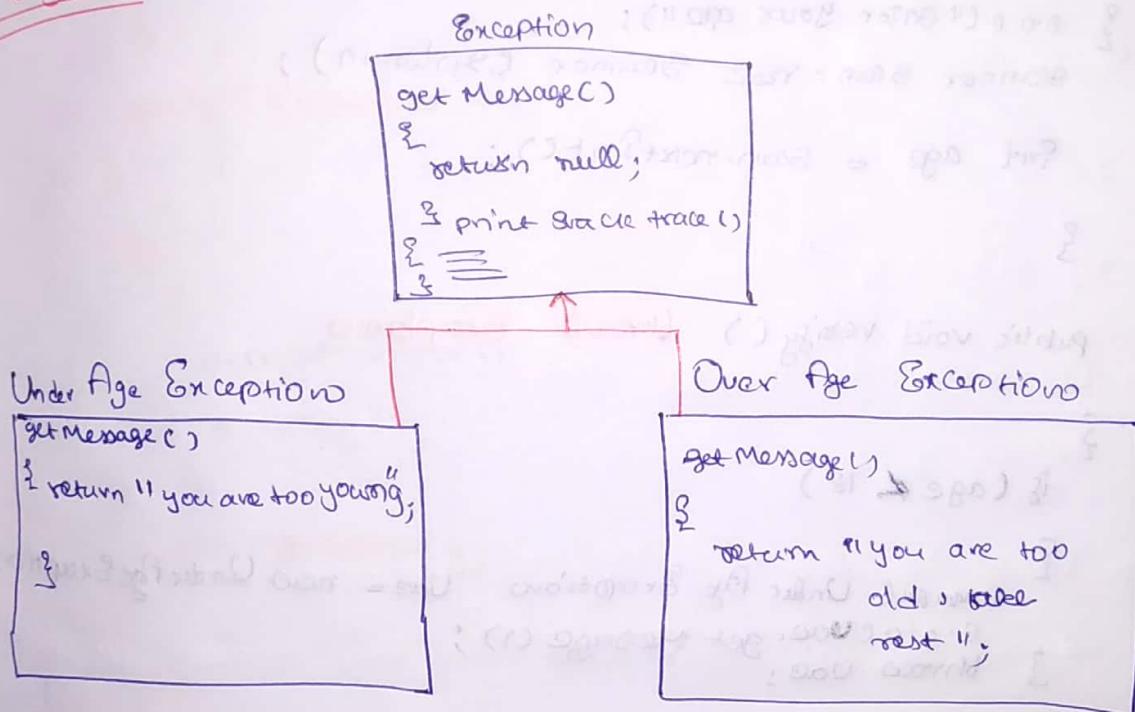
} }

class launch

{
 param (CString args[])

{
 Bank b = new Bank();
 b.initiate();
}

Program



class UnderAgeException extends Exception.

{
 Public String getMessage()
 {
 return "you are too young";
 }
}

class OverAgeException extends Exception

{
 Public String getMessage()
}

Yetano "you are too old, take rest";

۹۳

class Applicant

{ Private life age ;

```
public void acceptInput()
```

{ S.O.P ("Enter Your age");
Scanner scan = new Scanner (System.in);

~~int age = scan.nextInt();~~

۳

public void verify() throws Exceptions

§

To (age 18)

♀

Example Under Age Exception: var = new UnderAgeException("Sorry")

S.O.P.C Use. get message. (1)

3 throws Vac; get Message (1);

else if (age > 65)

8

OverAgeException Oae = new Over Age Exception();

SOP (One - get Message C.)

through One,

۲

else

2

8.0. problem 4 Eligible

9

to license "J";

3

class RTO

Ramana 20

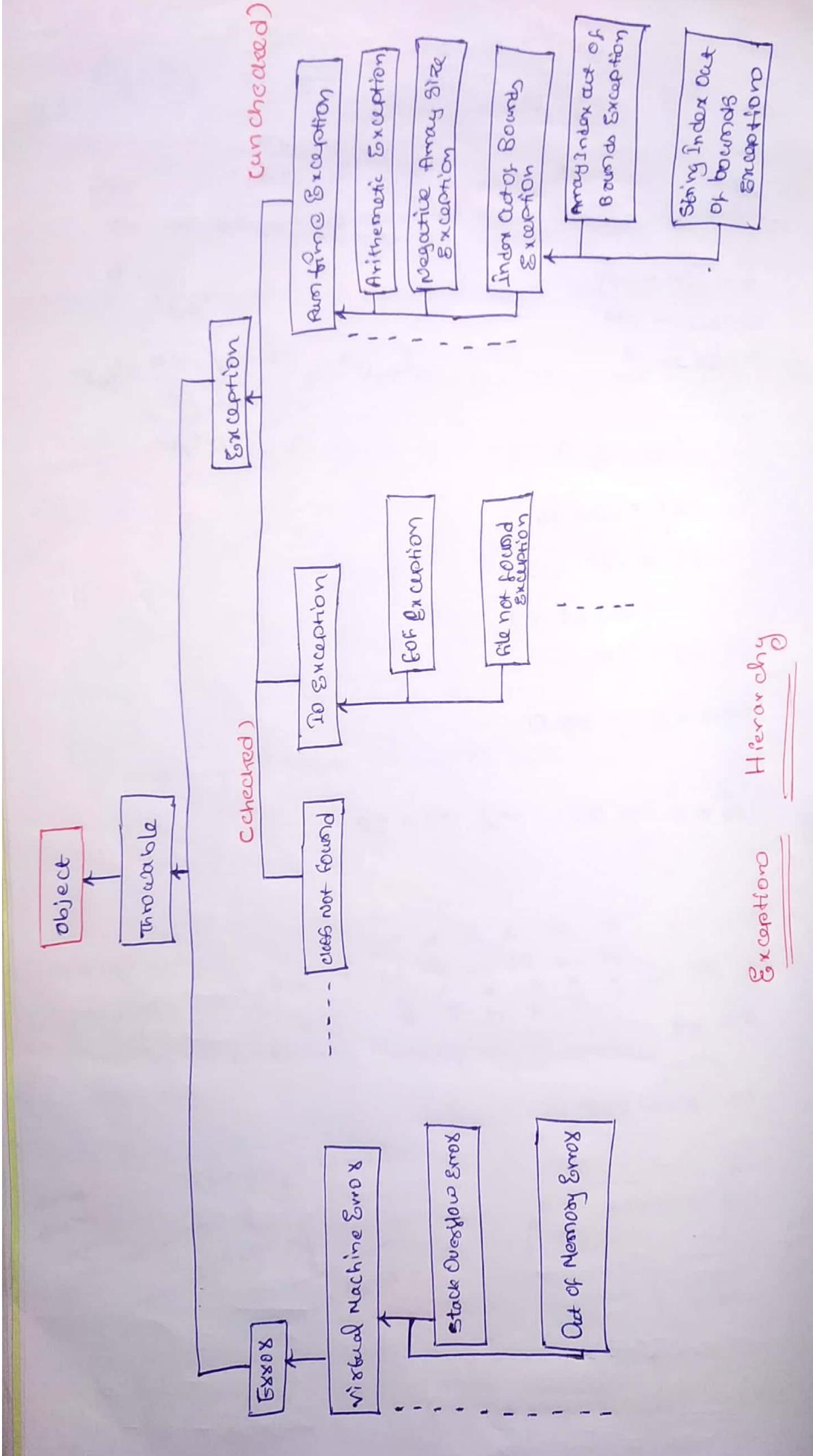
```
{  
    public void Permit() {  
        Applicant a = new Applicant();  
  
        try {  
            a.acceptInput();  
            a.verify();  
        } catch (Exception e) {  
            System.out.println("An error occurred while processing the application");  
            e.printStackTrace();  
        }  
        catch (Exception g) {  
            System.out.println("An error occurred while processing the application");  
            g.printStackTrace();  
        }  
        catch (Exception h) {  
            System.out.println("An error occurred while processing the application");  
            h.printStackTrace();  
        }  
    }  
}
```

Class Launch

```
{  
    public static void main (String args [ ] )  
    {  
        RTO x = new RTO ();  
        x . permit ();  
    }  
}
```

Steps involved in custom exception

1. Create a regular java class, make it an exception class by extending "Exception".
2. Override getMessage() and include the required error message inside getMessage().
3. Create the object of the exception class where it is required, call the getMessage() and print the message later throw the exception.
4. Handle the thrown exception in caller method. Call the exception generating method required number of times based on the application and use try-catch block to handle the exception every time this method is called.



checked exception :-

checked exceptions are such exceptions which the compiler forces the programmes to handle. Irrespective of whether the exception occurs or not, a checked exception has to be handled using a try-catch (or) has to be declared using throws, it can also be rethrown using throw.

unchecked exception :-

unchecked exceptions are such exceptions which are not forced by the compiler to be handled. If the programme is aware of the exception that might occur he has an option to handle it or not. However it is not an compulsion that it has to be handled.

Throw

- It is used to re-throw the handled exception.
- It is present within the body of the method.

Throws

- It is used to declare a checked exception.
- It is associated with the method signature.

LSP (Liskov Substitution Principle)

Rule-1

The child class overridden methods has to throw the same exception which that is being thrown by the overridden parent class method (or) child class method can avoid throwing an exception. However child class overridden method throwing a different exception where compare to parent class method is not possible.

illegal

class Demo

{

 public void disp() throws IOException

}

{

class Demo2 extends Demo

{

 public void disp() throws SQLException

{

{

is runtime exception then the child class overridden method can throw a different exception. But if the common parent is not runtime then the child class overridden method can't throw different exception.

legal

class Demo

{

 public void disp() throws ArithmeticException

{

}

}

class Demo2 extends Demo

{

 public void disp() throws NullPointerException

{

}

}

illegal

class Demo

{

 public void disp() throws EOFException

{

}

}

class Demo2 extends Demo

{

 public void disp() throws FileNotFoundException

{

}

}

Legal

class Demo

```
{ public void disp() throws IOException }
```

```
{ }
```

```
}
```

class Demo2 extends Demo

```
{ public void disp() throws IOException }
```

```
{ }
```

```
}
```

```
}
```

(or) public void disp()

```
{ }
```

```
}
```

Rule-2

The child class overridden method can throw a different exception if there is a Is-A relationship (or) extends relationship b/w the exceptions. However parent class method should throw parent type exception & child class overridden method should throw child type exception.
The vice-versa is illegal.

Illegal

class Demo

```
{ public void disp() throws IOException }
```

```
{ }
```

```
{ }
```

```
}
```

class Demo2 extends Demo

{
 public void disp() throws Exception

{

}

}

Legal

class Demo

{

 public void disp() throws Exception

{

 // body of disp method

}

} // ends defining a new disp method with ref overridden

class Demo2 extends Demo

{

 public void disp() throws IOException

{

}

}

Rule-3

If the exceptions thrown by the parent

class method and the child class overridden method

share a common parent and if the parent

is runtime exception then the child class overridden method can throw a different exception. But if the common parent is not run time then the child class overridden method can't throw different exception.

legal

```
class Demo  
{  
    public void disp() throws ArithmeticException  
}
```

```
class Demo2 extends Demo  
{  
    public void disp() throws NullPointerException  
}
```

illegal

```
class Demo  
{  
    public void disp() throws EOFException  
}
```

```
class Demo2 extends Demo  
{  
    public void disp() throws FileNotFoundException  
}
```

Rule - 4

The parent class method can throw any exception if the child class overridden method is throwing run-time exception (or) in other words irrespective of the exception which the parent class method is throwing child class overridden method can throw any run-time exception.

legal

class Demo

{

 Public void disp() throws SQLException, IOException, ClassNotFoundException

{

}

}

class Demo2 extends Demo

{

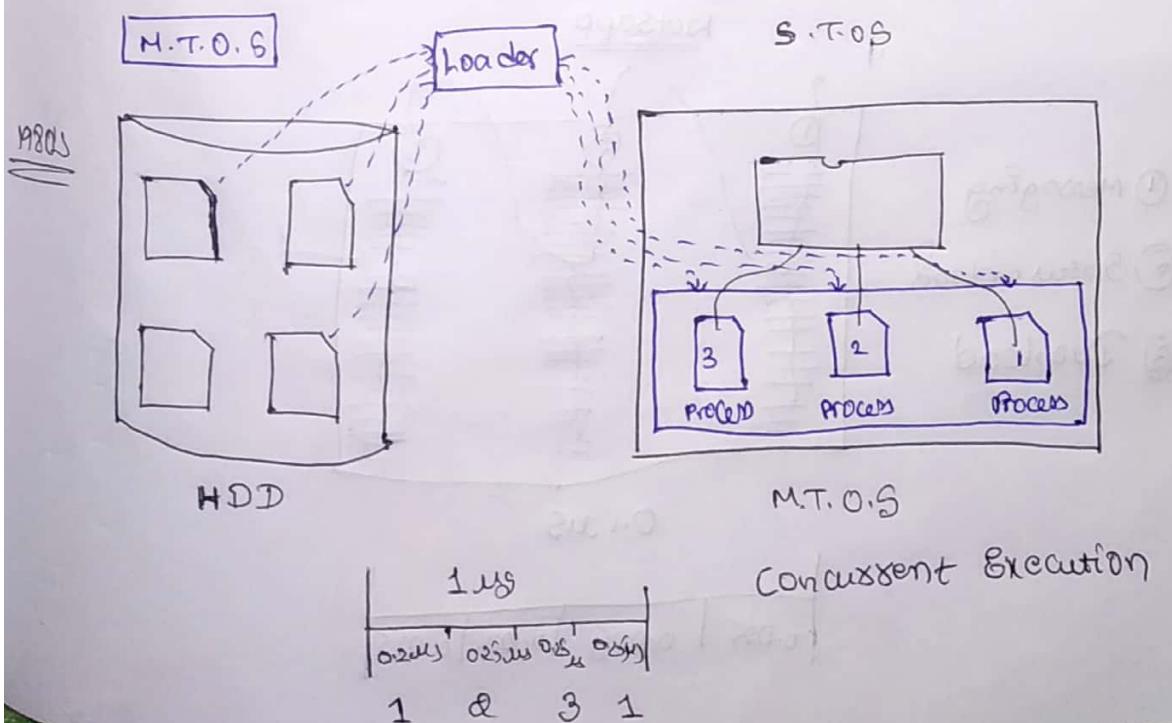
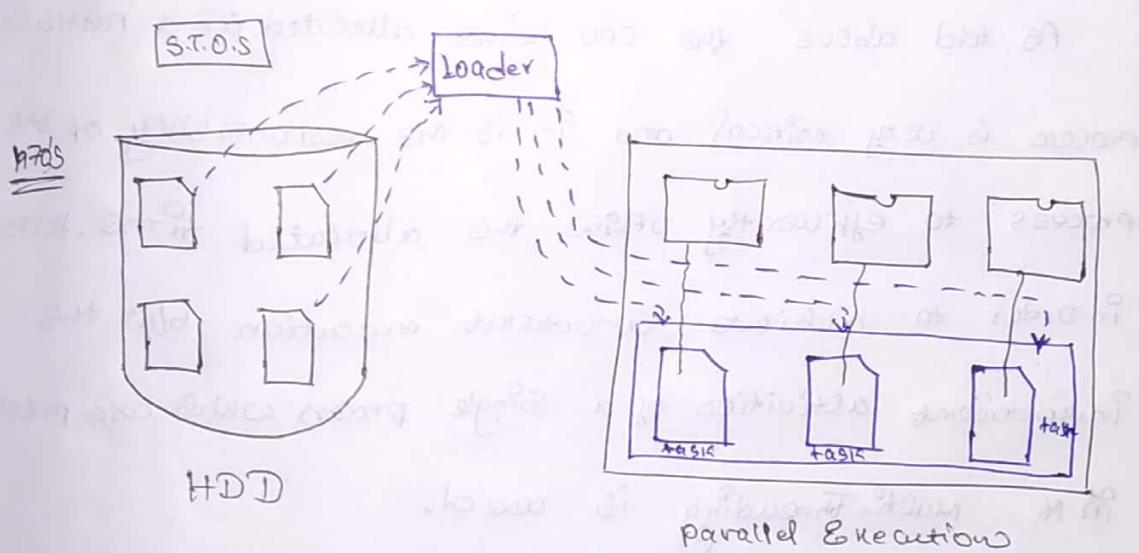
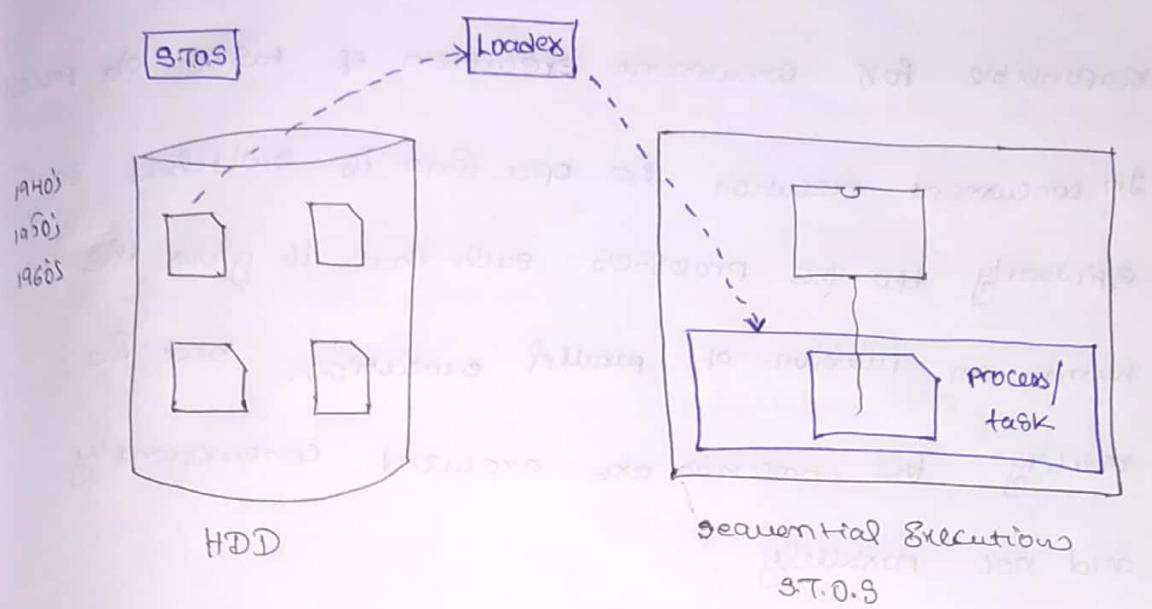
 Public void disp() throws Runtime Exception

{

}

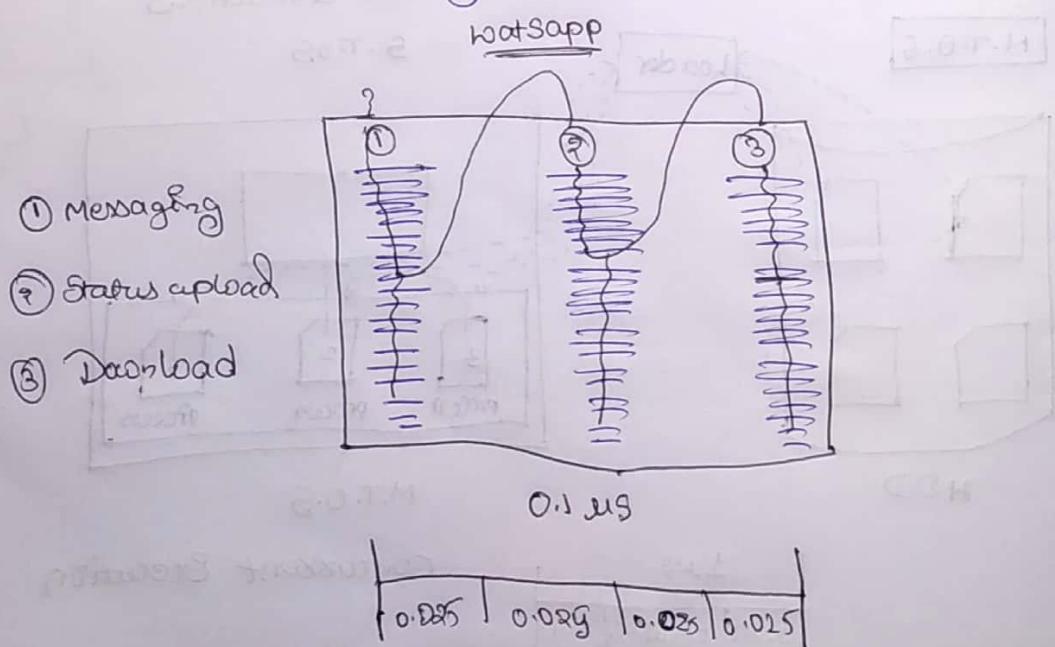
}

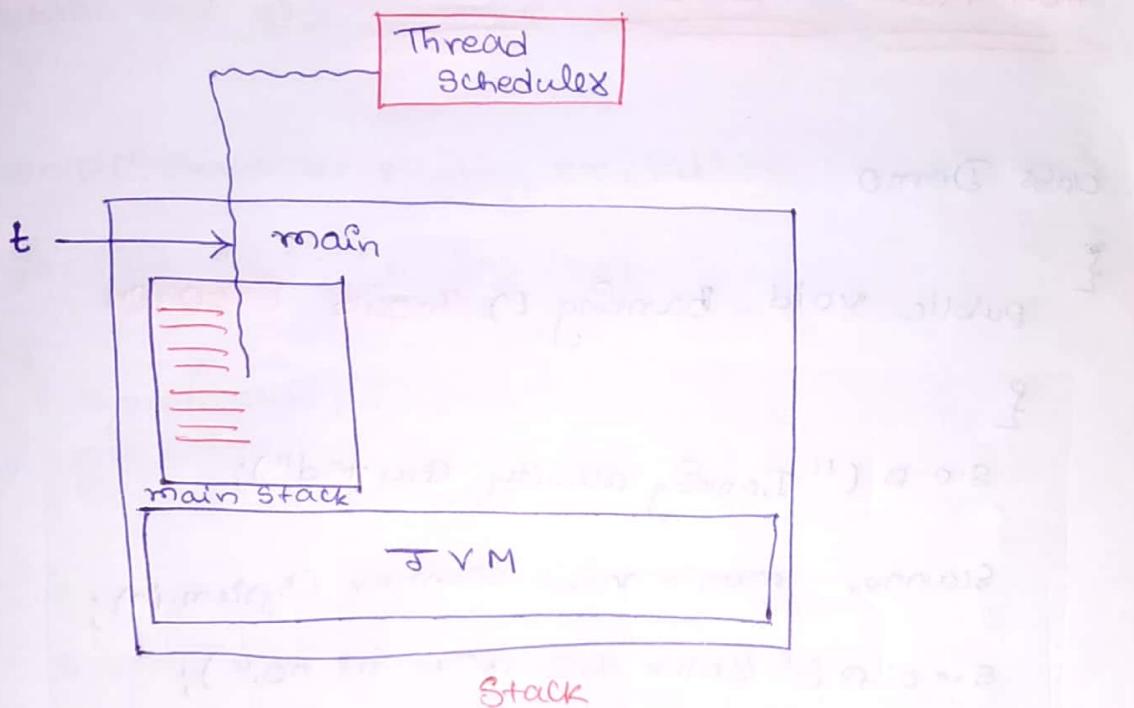
Multi-Threading



In order to efficiently utilise the CPU time between the processes, multitasking OS is required. M.T.O.S is responsible for concurrent execution of tasks of processes. In concurrent execution the CPU time is switched b/w the processes such that it gives the illusion of parallel execution. In reality the processes are executed sequentially and not parallelly.

As told above the CPU time allocated for a particular process is very critical and it is the responsibility of the process to efficiently utilise the allocated time. While in order to achieve concurrent execution b/w the independent activities of a single process which are present in it, multi-threading is used.





```

class Launch
{
    public static void main (String args[])
    {
        Thread t = Thread.currentThread();
        System.out.println(t); // Thread [main, 5, main]
    }
}

class Launch
{
    public static void main (String args)
    {
        Thread t = Thread.currentThread();
        t.setName("ABC");
        t.setPriority(1);
        System.out.println(t); // Thread [ABC, 1, main]
    }
}

```

name of Thread
priority of Thread
method under execution

Non Multi-Threaded Approach

Class Demo

{

 public void banking() throws Exception

{

 System.out.println("Banking activity started");

 Scanner scan = new Scanner(System.in);

 System.out.println("Enter the account no.");

 int acc = scan.nextInt();

 System.out.println("Enter the password");

 int ps = scan.nextInt();

 Thread.sleep(5000);

 System.out.println("Collect your money");

 System.out.println("Banking activity completed");

}

 public void numPrinting() throws Exception

{

 System.out.println("Number printing Started");

 for (int i = 1; i <= 5; i++)

 System.out.println(i);

 Thread.sleep(3000);

}

 System.out.println("Number printing completed");

}

public void charPrinting() throws Exception

{

s.o.p("Character printing started");

for (int i=65; i<=90; i++)

{

s.o.p((char) i);

Thread.sleep(3000);

}.

s.o.p("Character printing completed");

}

}

class Launch

{

public static void main(String args[]) throws Exception,

{

Demo d = new Demo();

d.longPrinting();

d.numPrinting();

d.charPrinting();

}

}

Multi-Threaded Approach

```
class Demo1 extends Thread  
{  
    public void run()  
    {  
        System.out.println("Banking activity started");  
        Scanner scan = new Scanner(System.in);  
        System.out.println("Enter acc-no");  
        int acc_no = scan.nextInt();  
        System.out.println("Enter password");  
        int psw = scan.nextInt();  
        Thread.sleep(5000);  
        System.out.println("Collect your money");  
        System.out.println("Banking activity completed");  
    }  
}
```

```
catch (Exception e)  
{  
    System.out.println("Some problem occurred");  
}
```

```
class Demo2 extends Thread  
{  
    public void run()  
    {  
        System.out.println("Banking activity started");  
    }  
}
```

```
S.o.p ("Number Printing Started");
for (int i = 0; i <= 5; i++)
{
    S.o.p(i);
    Thread.sleep(3000);
}
S.o.p ("Number Printing Completed");
}
```

class Demo3 extends Thread

```
{ public void run()
```

```
{ try
```

```
{
```

```
S.o.p ("Character printing started");
for (int i = 65; i <= 69; i++)

```

```
{
```

```
S.o.p ((char)i);
```

```
Thread.sleep(3000);
}
```

```
{
```

```
S.o.p ("Character printing completed");
}
```

```
{ catch (Exception e)
```

```
{ S.o.p ("Some problem occurred");
}
```

```
{ }
```

class Launch

{

 main (String args[])

{

 Demo1 d1 = new Demo1();

 Demo2 d2 = new Demo2();

 Demo3 d3 = new Demo3();

 d1.start();

 d2.start();

 d3.start();

}

}

class Launch

{

 main (String args[])

{

 Demo1 d1 = new Demo1();

 Demo2 d2 = new Demo2();

 Demo3 d3 = new Demo3();

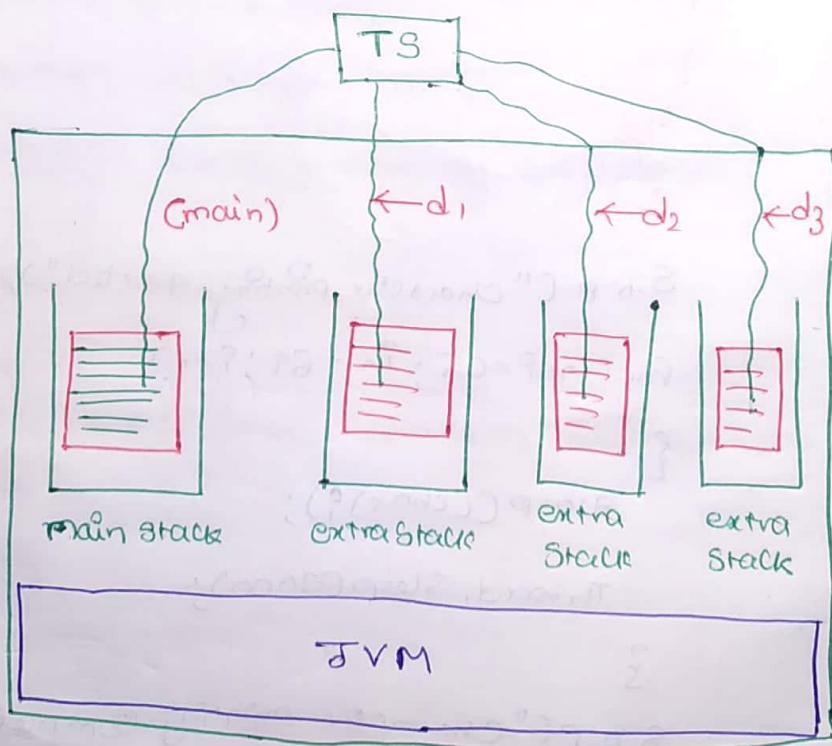
 d1.start();

 d2.start();

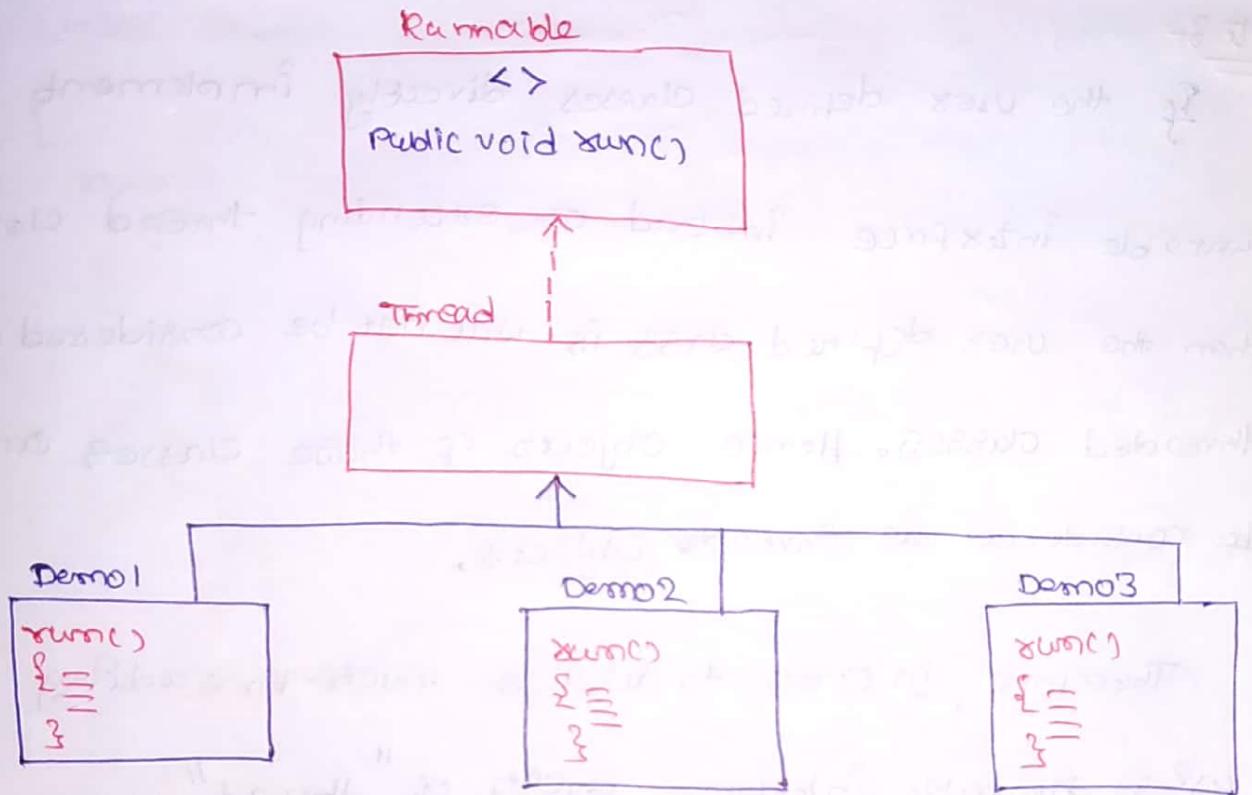
 d3.start();

}

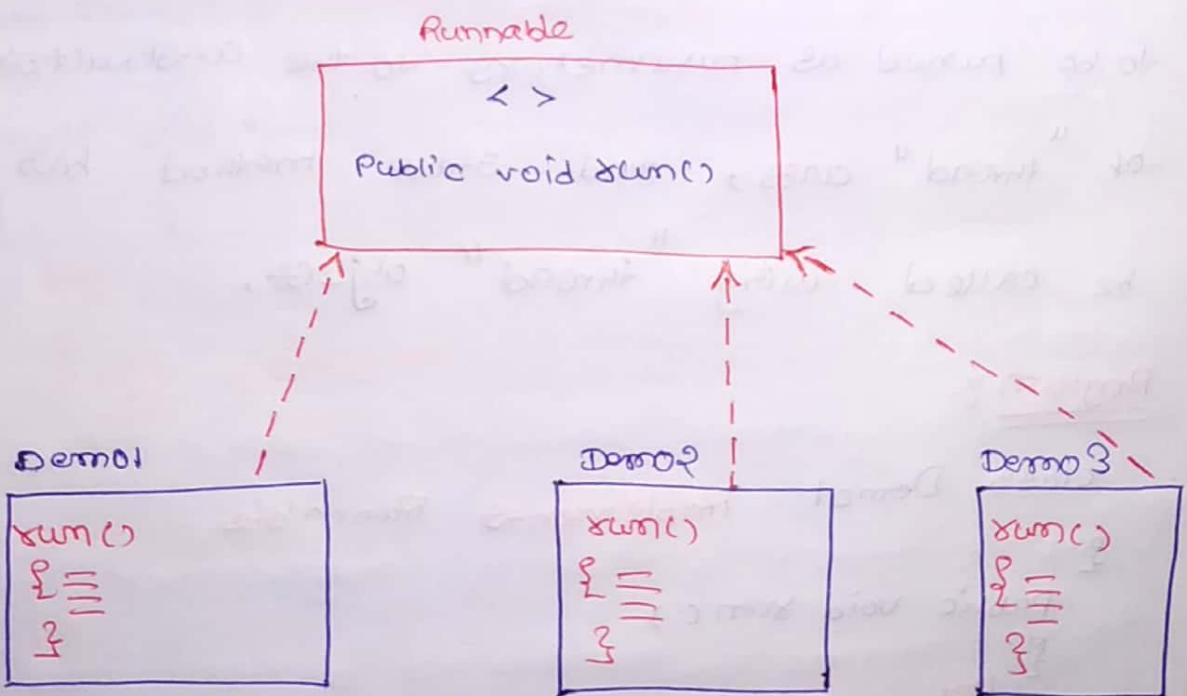
}



STACK



* Achieving Multi-Threading using Runnable Interface.



Note :-

If the user defined classes directly implement Runnable interface instead of extending thread class then the user defined class is will not be considered as threaded classes. Hence objects of these classes can't be considered as thread objects.

Therefore, in order to achieve multi-threading using runnable interface objects of "thread" class (in built java class) have to be created and references of user defined class objects have to be passed as parameters to the constructor of "thread" class, and start method has to be called using "thread" object.

Program :-

```
class Demo1 implements Runnable
```

```
{ public void run()
```

```
{ try
```

```
{ == (Running)
```

```
 }
```

```
 catch (Exception e)
```

```
{ ==
```

```
} }
```

```
class Demo2 implements Runnable  
{  
    public void run()  
{  
        try  
        {  
            // (Printing)  
        }  
        catch (Exception e)  
        {  
        }  
    }  
}
```

```
class Demo3 implements Runnable
```

```
{  
    public void run()  
{  
        try  
        {  
            // (Character  
            // Printing)  
        }  
    }  
}
```

```
catch (Exception e)
```

```
{  
}
```

```
{  
}
```

```
class Launch
```

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

Demol1 d1 = new Demol();

Demol2 d2 = new Demol2();

Demol3 d3 = new Demol3();

Thread t1 = new Thread(d1);

Thread t2 = new Thread(d2);

Thread t3 = new Thread(d3);

t1.start();

t2.start();

t3.start();

}

Note :-

A programmer should never explicitly call "run()"

becoz if run() is called by programmer then
the run method is treated as a specialised method
of user defined class and its activation record
gets created on the main stack just how
the hooy how activation records of other
regular methods are created.

If start method calls the run() then
the activation record of run method is not created
on the main stack. but it is created on ~~stack~~
extra stack hence concurrent execution
can be achieved using extra threads.

If the programmer calls run() it results in
sequential execution. Hence in order to avoid -
sequential execution and achieve concurrent
execution run method has to be called using
start method.

class Demo1 extends Thread

{

 public void run()

{

 try

 {

 (Bombing)

}

 catch (Exception)

{

=

}

}

class Demo2 extends Thread

```
{ public void run()
```

```
{
```

```
try
```

```
{ == (number printing)
```

```
{
```

```
catch (Exception e)
```

```
{ ==
```

```
{
```

```
}
```

class Demo3 extends Thread

```
{ public void run()
```

```
{
```

```
try
```

```
{ == (character  
printing)
```

```
catch (Exception e)
```

```
{ ==
```

```
{
```

```
}
```

class Launch

```
{
```

```
public static void main (String args [])
```

```
{
```

```
Demol d1 = new Demol();
```

```
Demol d2 = new Demol();
```

```
Demol d3 = new Demol();
```

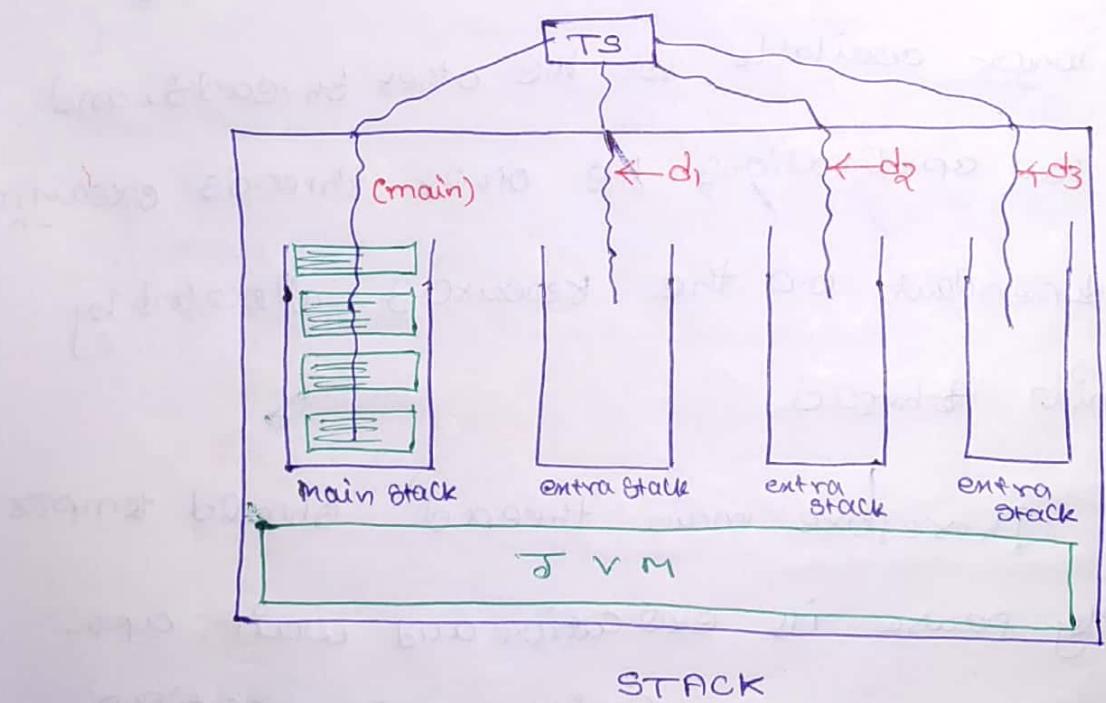
```
d1.fun();
```

```
d2.fun();
```

```
d3.fun();
```

```
}
```

```
}
```



Join() and isAlive()

Main thread is the first thread to start its execution and it is the first to complete execution. However the resources allocated by main thread will be available as long as main thread is alive. Main thread is moved to dead state once the execution is complete. Hence once main thread is moved to dead state the resources are no longer available for the other threads. In few applications the other threads execution is dependent on the resources allocated by main thread.

Therefore main thread should temporarily pause its execution and wait upon till other threads finish their executions. This can be achieved by using Join(). isAlive() is used to check whether the thread is still alive or not.

Program

```
class Demo1 implements Runnable
```

```
{  
    public void run()
```

```
{  
    try  
    {  
        == [Banking]  
    }  
}
```

Catch (Exception e)

```
{  
    ==  
}  
}  
}
```

```
class Demo2 implements Runnable
```

```
{  
    public void run()
```

```
{  
    try (Num Printing)  
    {  
        ==  
    }  
}
```

Catch (Exception e)

```
{  
    ==  
}
```

```
{  
}
```

class Demo3 implements Runnable

```
{  
    public void run()  
{  
        try  
{  
            == (char printing)  
        }  
    }  
}
```

catch (Exception e)

```
{  
    ==  
    ==  
}  
}
```

class Launch

```
{  
    public static void main (String args[]) throws Exception  
{  
        System.out.println ("main thread started its execution allocated  
        resources");  
    }  
}
```

Demo1 d1 = new Demo1();

Demo2 d2 = new Demo2();

Demo3 d3 = new Demo3();

Thread t1 = new Thread (d1);

Thread t2 = new Thread (d2);

Thread t3 = new Thread (d3);

System.out.println (t1.isAlive()); //false

System.out.println (t2.isAlive()); //false

System.out.println (t3.isAlive()); //false

t₁.start();

t₂.start();

t₃.start();

s.o.println(t₁.isAlive()); // true

s.o.println(t₂.isAlive()); // true

s.o.println(t₃.isAlive()); // true

t₁.join();

t₂.join();

t₃.join();

s.o.println(t₁.isAlive()); // false

s.o.println(t₂.isAlive()); // false

s.o.println(t₃.isAlive()); // false

s.o.println("main thread completed its execution de-

allocated resources");

}

}

Achieving multi-Threading using single run method

class Demo implements Runnable

{
 public void run()

{
 Thread t = Thread.currentThread();

 String name = t.getName();

 if (name.equals ("BANK"))

{

 banking();

}

 else if (name.equals ("NUM"))

{

 numPrinting();

}

 else

{

 char Printing();

}

 public void banking()

{

 try

{

 S.o.P (" Banking activity Started");

 Scanner Scan = new Scanner (System.in);

 S.o.P (" Enter acc-no ");

 int a = Scan.nextInt();

```
s.o.p ("Enter password");
```

```
int pswd = Scan.nextInt();
```

```
Thread.sleep(5000);
```

```
s.o.p ("Collect your money");
```

```
s.o.p ("Banking activity completed");
```

```
}
```

```
Catch (Exception e)
```

```
{
```

```
s.o.p ("Some problem occurred");
```

```
}
```

```
public void numPrinting()
```

```
{ try
```

```
{ s.o.p ("Number printing started");
```

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

```
{ s.o.p (i);
```

```
Thread.sleep(3000);
```

```
}
```

```
s.o.p ("Number printing completed");
```

```
Catch (Exception e)
```

```
{
```

```
s.o.p ("Some problem occurred");
```

```
}
```

```
}
```

```
public void charPrinting()
```

```
{
```

```
try
```

```
{
```

```
System.out.println("Character printing started");
```

```
for (int i = 65; i <= 69; i++)
```

```
{
```

```
System.out.print((char) i);
```

```
Thread.sleep(3000);
```

```
}
```

```
System.out.println("Character printing completed");
```

```
}
```

```
catch (Exception e)
```

```
{
```

```
System.out.println("Some problem occurred");
```

```
}
```

```
{
```

```
} public void main
```

```
{ class Launch
```

```
public static void main(String args[])
```

```
{
```

```
Demo d = new Demo();
```

```
Thread t1 = new Thread(d);
```

```
Thread t2 = new Thread(d);
```

```
Thread t3 = new Thread(d);
```

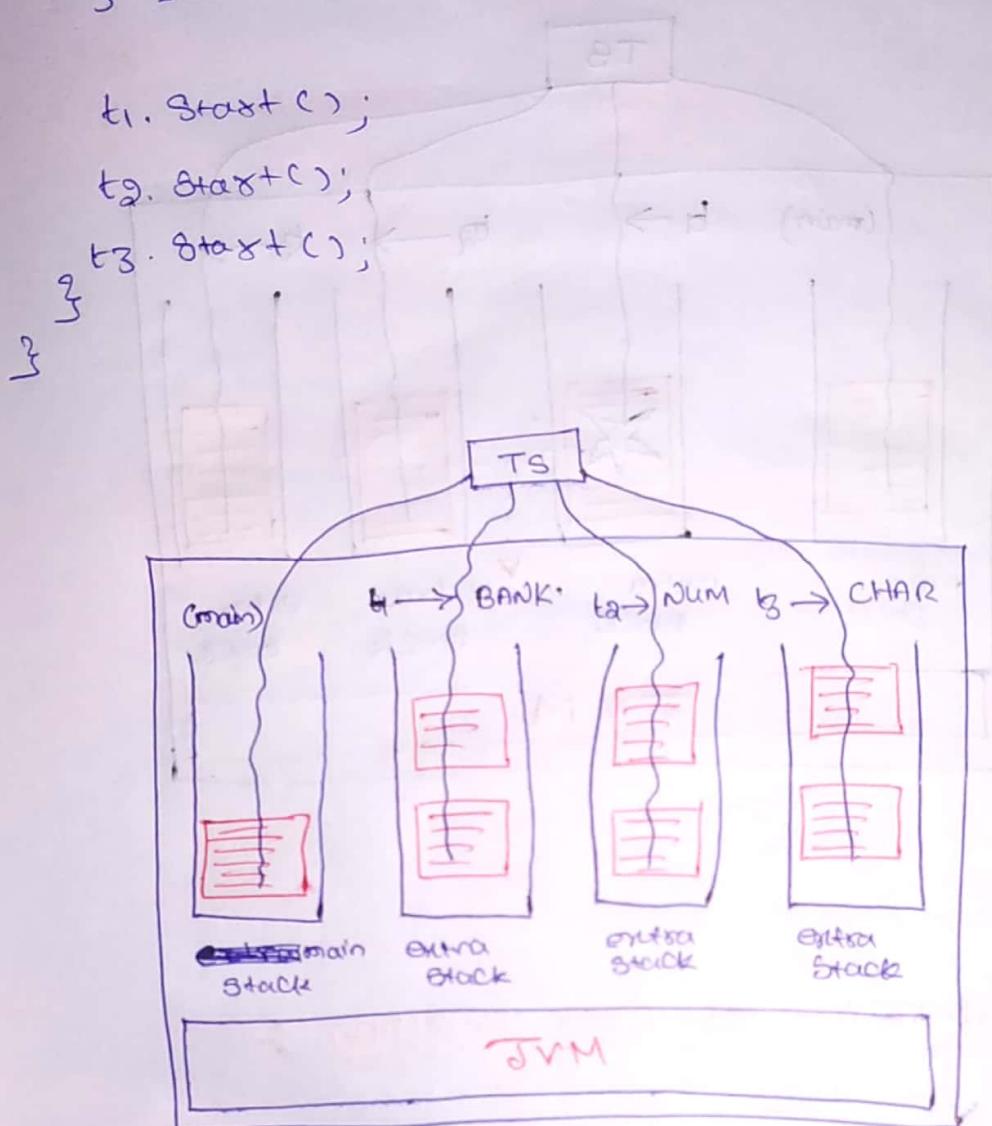
t₁. setName ("BANK");
t₂. setName ("NUM");
t₃. setName ("CHAR");

t₁. start();
t₂. start();
t₃. start();
}

t₁. setName ("BANK");

t₂. setName ("NUM");

t₃. setName ("CHAR");

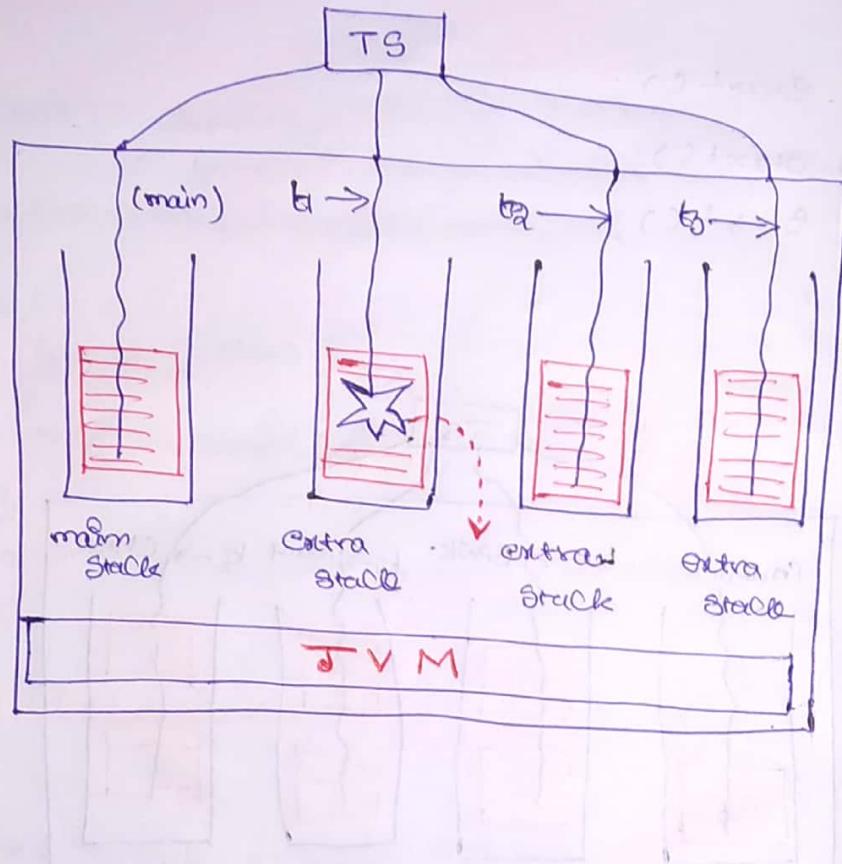


Exception in Multi-Threading

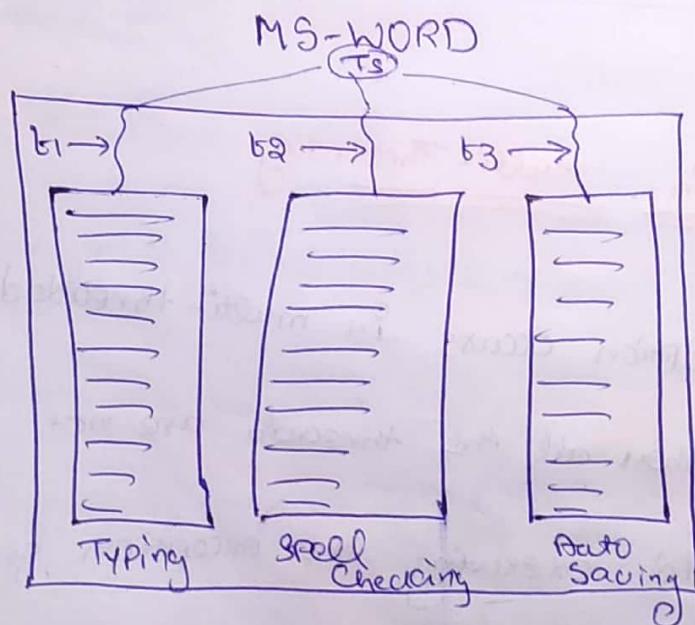
* If an exception occurs in multi-threaded environment then all the threads are not affected.

The thread executing the exception generating method will stop its execution but however

The other threads will continue as normal.



Demonstration of Race Condition :-



```
class MyWord extends Thread  
{  
    public void run()  
{  
        try  
{  
            Thread t = Thread.currentThread();  
            String name = t.getName();  
            if (name.equals("TYPE"))  
            {  
                typing();  
            }  
            else if (name.equals("SPELL"))  
            {  
                spellChecking();  
            }  
            else  
            {  
                autoSaving();  
            }  
        }  
        catch (Exception e)  
        {  
            System.out.println("Some problem occurred");  
        }  
    }  
}
```

public void typing() throws Exception

```
{
```

for (int i=1; i<=5; i++)

```
{
```

//Keyboard input = 100ms
 //Keyboard input = 80ms

s.o.p("bying ----");

Thread.sleep(3000);

}

}

public void spellChecking() throws Exception

{

for (int i=1; i<=5; i++)

{

s.o.p("spell-checking");

Thread.sleep(3000);

}

}

public void autoSaving() throws Exception

{

for (int i=1; i<=5; i++)

{

s.o.p("auto-saving - - - -");

Thread.sleep(3000);

}

}

}

class Launch

{

psvm (String args[]) throws Exception

{

MSWord msg1 = new MSWord();

MSWord msg2 = new MSWord();

MSWord msg3 = new MSWord();

```
msg1.setName("TYPE");
```

```
msg2.setName("SPELL");
```

```
msg3.setName("SAVE");
```

```
ms1.start();
```

```
ms2.start();
```

```
msg.start();
```

}

}

→ In the shown program we can encounter race condition because there are multiple activities each executed by a particular thread and every thread attempts to finish its execution as soon as possible. Due to this the output of the program can't be predicted and the behaviour of the software can be considered inconsistent. This effects the performance of the software. Hence higher priority has to be given main activities and less priority has given to the sub-ordinate or secondary activities. This can be achieved using Daemon Threads.

Steps to Create Daemon Threads

1. Identify Secondary activities.
2. Include secondary activity inside a infinite loop.
3. Set the setDaemon() as true using threads which will execute secondary activity.
4. Assign low priority to daemon threads.

Program

```
class MSWord extends Thread  
{  
    public void run()  
    {  
        try  
        {  
            Thread t = Thread.currentThread();  
            String name = t.getName();  
            if (name.equals("TYPE"))  
            {  
                typing();  
            }  
            else if (name.equals("SPELL"))  
            {  
                spellChecking();  
            }  
            else  
            {  
                autoSaving();  
            }  
        }  
    }  
}
```

catch (Exception e)

{

s.o.p ("Some problem occurred");

}

}

public void typing() throws Exception

{

for (int i=1; i<=5; i++)

{

s.o.p ("typing ----");

Thread.sleep(3000);

}

}

public void spellChecking() throws Exception

{

for(;;)

{

s.o.p ("spell checking ----");

Thread.sleep(3000);

}

}

public void autoSaving() throws Exception

{

for(;;)

{

s.o.p ("auto Saving ----");

Thread.sleep(3000);

}

}

}

Random access

class Launch

{

P S V m C String args[]) throws Exceptions

{

MSWord ms1 = new MSWord();

MSWord ms2 = new MSWord();

MSWord ms3 = new MSWord();

ms1.setName("TYPE");

ms2.setName("SPELL");

ms3.setName("SAVE");

ms2.setDaemon(true);

ms3.setDaemon(true);

ms2.setPriority(3);

ms3.setPriority(2);

ms1.start();

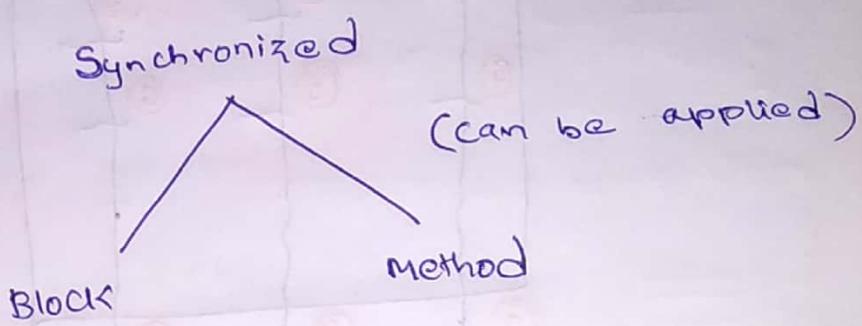
ms2.start();

ms3.start();

}

}

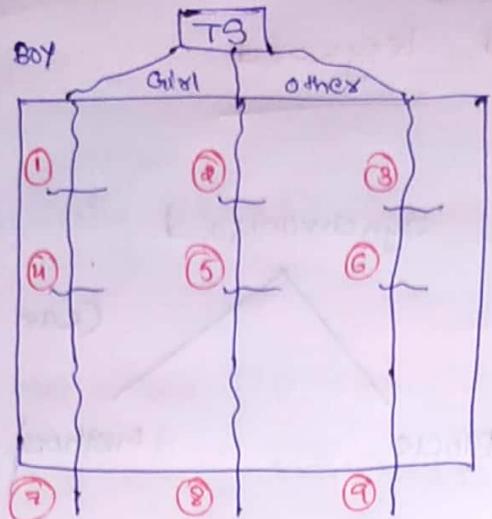
Synchronized keyword



when a common resource is being accessed by multiple threads if one thread wastes time opportunity (the) CPU time is given to another thread and the CPU time will keep switching and it is given to multiple threads In such a case none of the threads will be able to fully utilise the common resource completely. This can result in an unpredictable output and uncertain []

Usage of Common resource

Program



class Trainroom implements Runnable

{

 public void run()

{

 try

 {

 Thread t = Thread.currentThread();

 String name = t.getName();

 System.out.println(name + " has entered the trainroom");

 Thread.sleep(5000);

 System.out.println(name + " has left the trainroom");

 Thread.sleep(5000);

 System.out.println(name + " has exited the trainroom");

 }

 catch (Exception e)

{

 System.out.println("Trainroom activity interrupted");

}

}

class launch

{
 p.su.m casting args[])

}

Bathroom b = new Bathroom();

Thread t1 = new Thread(b);

Thread t2 = new Thread(b);

Thread t3 = new Thread(b);

t1.setName("BOY");

t2.setName("GIRL");

t3.setName("OTHER");

t1.start();

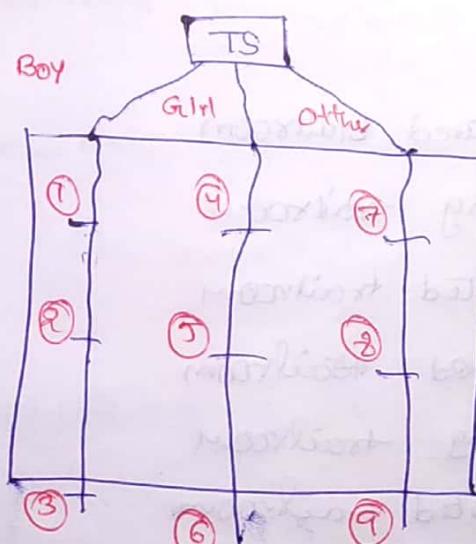
t2.start();

t3.start();

}

3

program



class Bathroom

implements Runnable

{

Synchronized public void run()

{

Boy

{

====

}

Catch (Exception e)

{

====

}

} }

class Launch

{

 public static void main(String args[])

{

====
====
====
====

} }

Output :-

Boy Entered trailroom

Boy Using trailroom

Boy Exited trailroom

Girl Entered trailroom

Girl Using trailroom

Girl Exited trailroom

Other entered trailroom

Other Using trailroom

Other Exited trailroom

Person who

Ques 8-
Semaphore (Ans) Monitor is nothing but
set of lines of code which is accessible and
executed by a single thread completely and
the semaphore can be executed by other thread
only once the other threads has completely finished
its execution.

(Q6)

Semaphore or monitor is a part of code which
can be executed by only a single thread at a
given time interval completely.

Synchronized block

class Demo implements Runnable

{

 public void run()

{

 try

 {

 for (int i=1; i<=5; i++)

 {

 S.O.P(i);

 Thread.sleep(3000);

 }

 synchronized (this)

```
for (int i = 65; i <= 69; i++)
```

```
{
```

```
    s.o.p(cchar)i;
```

```
}
```

```
Thread.sleep(3000);
```

↳ Thread.sleep() blocks update of screen

```
3
```

```
3
```

```
Catch CException e)
```

↳ catch block handles exception

```
{
```

```
s.o.p("some problem occurred");
```

```
3
```

```
3
```

```
class Demo
```

```
{
```

```
    public void main(String args[])
```

```
{
```

```
    Demo d = new Demo();
```

```
    Thread t1 = new Thread(d);
```

```
    Thread t2 = new Thread(d);
```

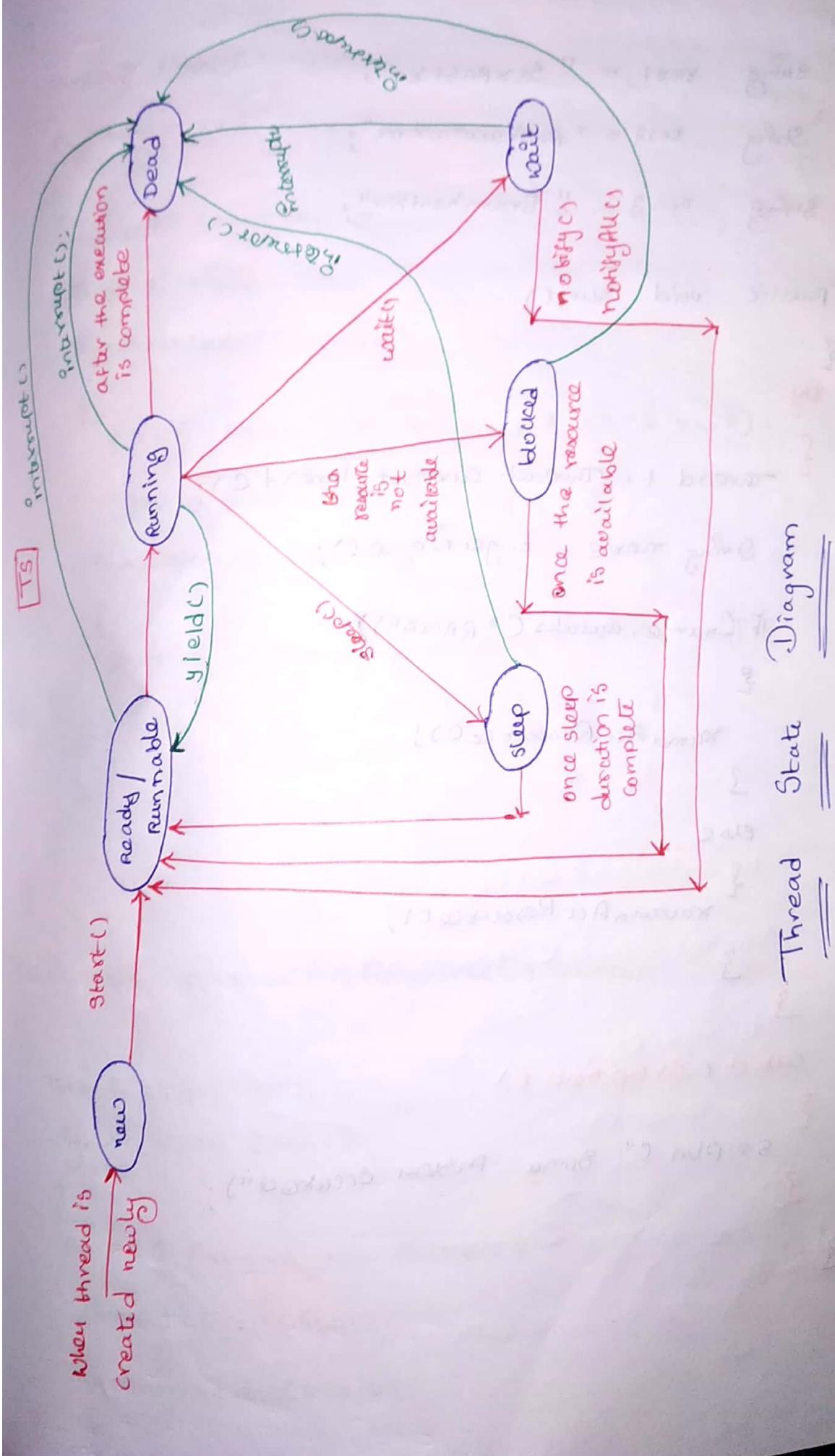
```
    t1.start();
```

```
    t2.start();
```

```
}
```

```
3
```

States of Thread :-



class Warrior implements Runnable

{

String res1 = "Saxpatalxa";

String res2 = "pashurastastrol";

String res3 = "Bramhastxal";

public void run()

{

try

{

Thread t = Thread.currentThread();

String name = t.getName();

If (name.equals("RANA"))

{

yamaAccResource();

}

else

{

savanaAccResource();

}

Catch (Exception e)

{

s.o.pln ("Some Problem occurred");

public void ramaAccResourceC() throws Exception

{
 Thread.sleep(5000);
 synchronized (res1){

{
 System.out.println("Rama has occupied " + " + res1");
 Thread.sleep(5000);
 synchronized (res2){

{
 System.out.println("Rama has occupied " + " + res2");
 Thread.sleep(5000);
 synchronized (res3){

{
 System.out.println("Rama has occupied " + " + res3");
}

{
 System.out.println("Rama has occupied " + " + res4");
}

{
 System.out.println("Rama has occupied " + " + res5");
}

public void ravanaAccResourceC() throws Exception

{
 Thread.sleep(5000);
 synchronized (res1){

{
 System.out.println("Ravana has occupied " + " + res1");
}

 Thread.sleep(5000);

 synchronized (res2){

{
 System.out.println("Ravana has occupied " + " + res2");
}

Thread.sleep(5000);

Synchronized (res3)

{

System.out.println("Ravana has occupied "+ " + res3);

}

{

{

{

{

class Launch

" "+ " between Ravana and Ram")

{

public void main (String args []) throws IOException

{

Warrior w = new Warrior();

Thread t1 = new Thread (w);

Thread t2 = new Thread (w);

t1.setName ("RAMA");

t2.setName ("RAVANA");

t1.start();

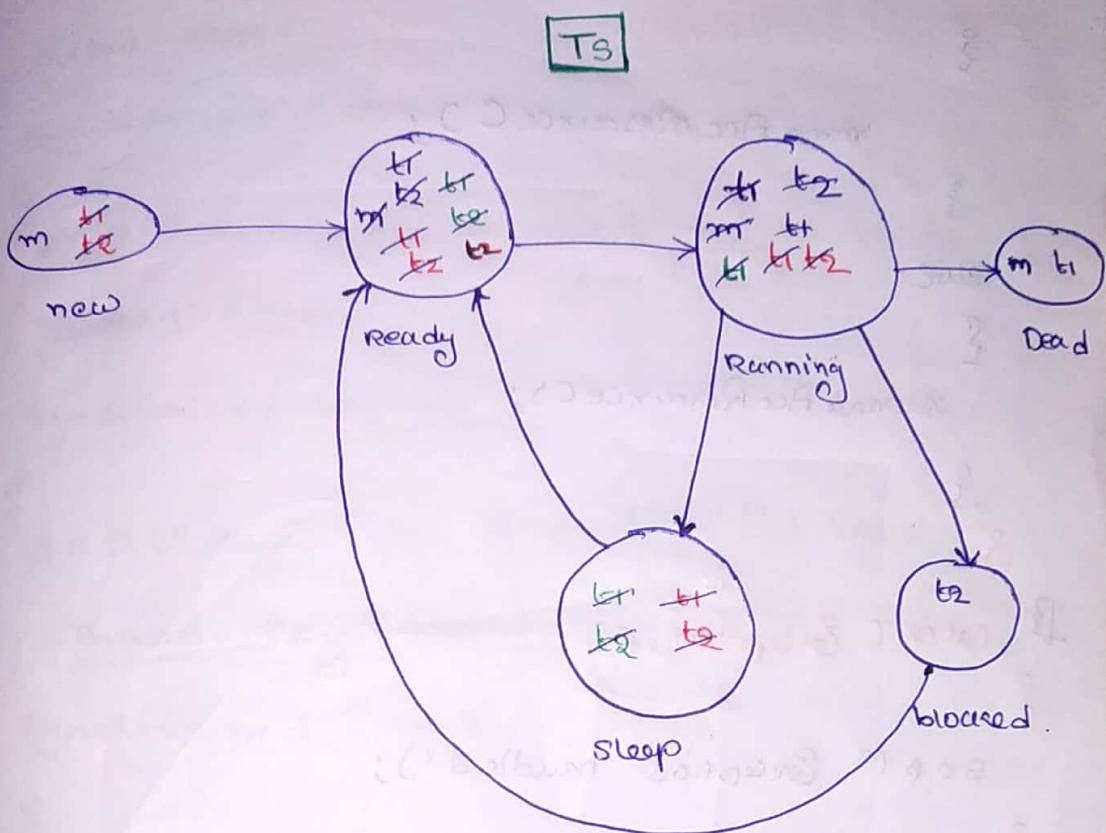
t2.start();

}

Thread
~~~~~

State  
~~~~~

Diagram
~~~~~



## Dead Lock Program

```
class Warrior implements Runnable
```

```
{
```

```
String sas = "Saxabattra";
```

```
String pas = "Pashupatabatra";
```

```
String brahma = "Brahmastra";
```

```
public void run()
```

```
{
```

```
try
```

```
{
```

```
Thread t = Thread.currentThread();
```

```
String name = t.getName();
```

if (name.equals("RAMA"))

{

Rama Acc Resource();

}

else

{

Savana Acc Resource();

}

3

try {

Catch (Exception e)

{

S.o.p ("Exception handled");

}

}

public void Rama Acc Resource() throws Exception

{

Thread. sleep(5000);

Synchronized (res1)

{

S.o.p ("Rama has Occupied" + " " + res1);

Thread. sleep(5000);

Synchronized (res2)

{ S.o.p ("Rama has Occupied" + " " + res2);

Thread. sleep(5000);

Synchronized (res3)

{

S.o.p ("Rama has Occupied" + " " + res3);

}

3  
3

```
public void ravanaAccResource() throws Exception
```

```
{
```

```
    Thread.sleep(5000);
```

```
    synchronized (res3)
```

```
{
```

```
        S.O.P("Ravana has occupied '+' "+ res3);
```

```
    Thread.sleep(5000);
```

```
    synchronized (res2)
```

```
{
```

```
        S.O.P("Ravana has occupied '+' "+ res2);
```

```
    Thread.sleep(5000);
```

```
    synchronized (res1)
```

```
{
```

```
        S.O.P("Ravana has occupied '+' "+ res1);
```

```
class Launch
```

```
{
```

```
    public static void main(String args[])
```

```
{
```

```
    warrior w = new Warrior();
```

```
    Thread t1 = new Thread(w);
```

```
    Thread t2 = new Thread(w);
```

```
    t1.setName("RAMA");
```

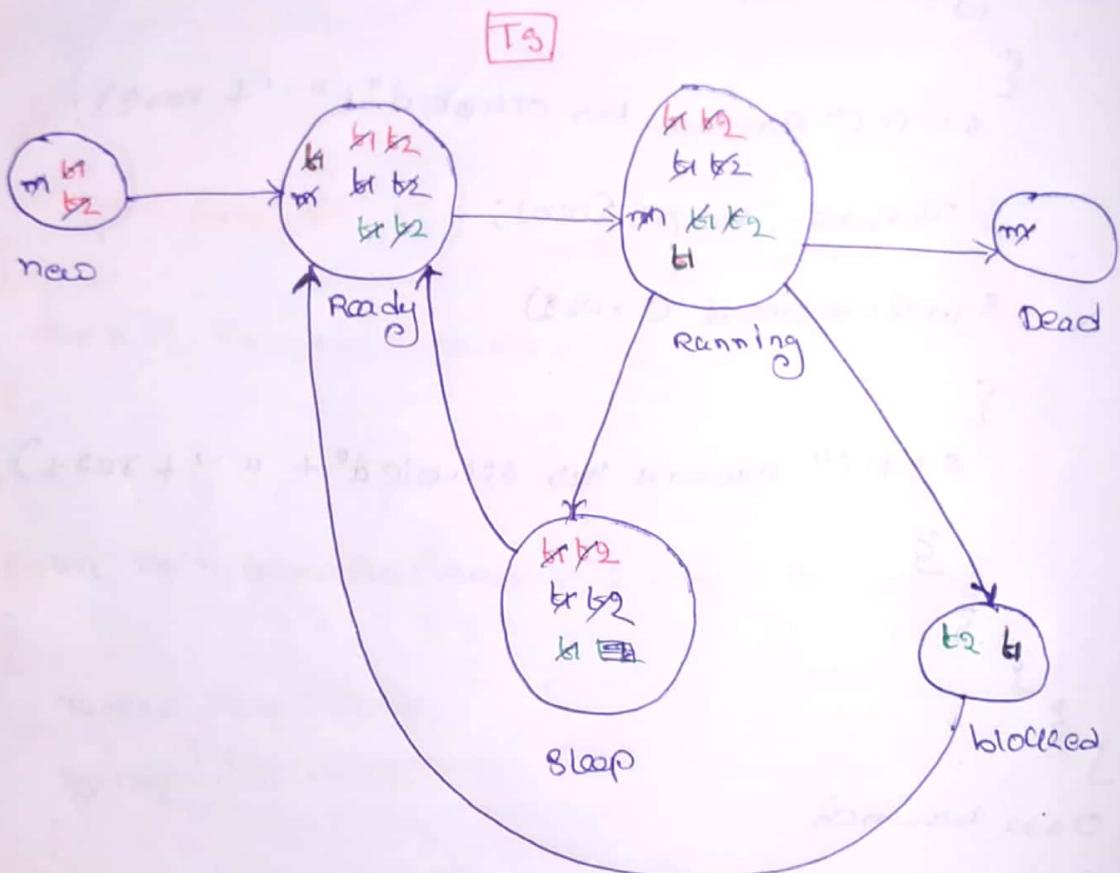
```
    t2.setName("RAVANA");
```

b, start();

t2.start();

3

3



## Threads

State

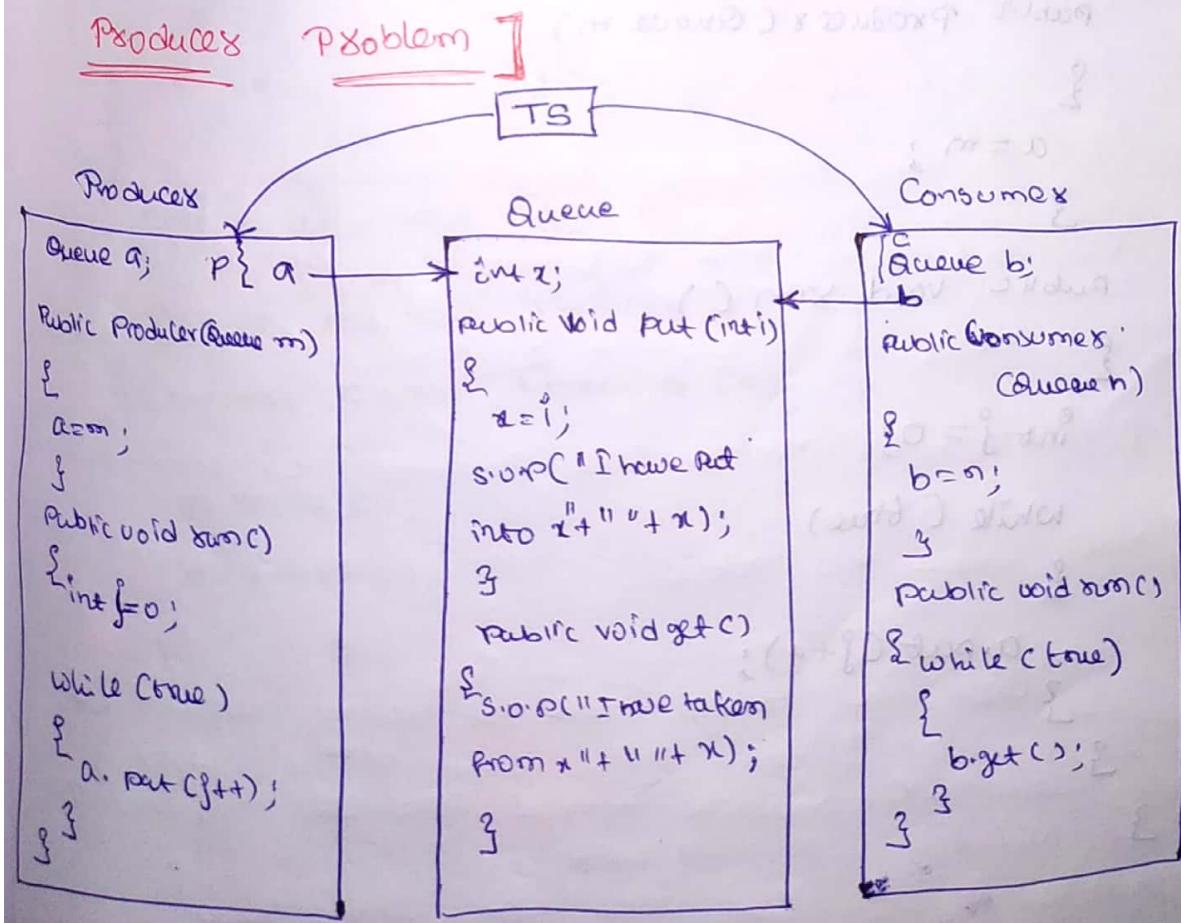
## Diagrams

Note

→ Deadlock is a phenomenon where multiple threads present in the block state fail to move to runnable state due to the cyclic dependency so mutual dependency that exists between them. Therefore we can say that a deadlock is created due to

"cyclic dependencies". Hence it is the responsibility of the programmer to write the code in such a way that deadlocks are not generated.

### Inter Communication between the threads [Consumer -



0 1, 2, 3, 4, 5 --- 99 --- 999

## Program

Class Queue

```
int x;  
public void put (int i)  
{  
    x = i;  
    System.out.println ("I have Put into x " + x);  
}
```

```
public void get ()  
{  
    System.out.println ("I have taken from x " + x);  
}
```

class Producer extends Thread

```
{ Queue a;  
public Producer (Queue m)  
{  
    a = m;
```

```
    public void run ()  
    {
```

```
        int j = 0;
```

```
        while (true)
```

```
        {
```

```
            a.put (j++);
```

```
        }
```

```
    }
```

```
}
```

```

class Consumer extends Thread
{
    Queue b;
    public Consumer (Queue n)
    {
        b = n;
    }
    public void run()
    {
        while (true)
        {
            b.get();
        }
    }
}

class launch
{
    public static void main (String args[])
    {
        Queue q = new Queue ();
        Producer p = new Producer (q);
        Consumer c = new Consumer (q);
        p.start ();
        c.start ();
    }
}

```

In the above program the threads created which are consumer thread and producer thread will enter an infinite loop. Hence the threads will never finish their execution. Therefore Thread scheduler switches and provides CPU time to both the threads alternatively even if there is no delay.

Once the CPU time is given to any of the threads they execute infinitely without time delay. If producer starts executing it will continuously drops values into 'x' without checking whether the consumer has consume the value or not. In other words -

Irrespective of whether the consumer consumed the value or not, producer keeps producing the values. and once CPU time is given to the consumer, it continuously consumes "latest" value irrespective of whether the producer has produced a new value or not.

i.e., the same value is being consumed by consumer multiple times and all the values produced by producer are not consumed.

This problem can be resolved by establishing inter common  
communications b/w the threads. Using "wait()" and "notify()".  
methods.

### consumer

- i) Consume the value
- ii) notify the producer
- iii) wait until producer produces a new value.

### producer

- i) produce the value
- ii) notify the consumer
- iii) wait until the consumer has consumed  
the new value.

## Program

```
class Queue  
{  
    int x;  
    boolean value_in_x = false;  
  
    synchronized  
    {  
        public void put(int i)  
        {  
            try  
            {  
                if (value_in_x == true)  
                {  
                    wait();  
                }  
                else  
                {  
                    x = i;  
                    System.out.println("I have put into " + x);  
                    value_in_x = true;  
                    notify();  
                }  
            }  
            catch (Exception e)  
            {  
                e.printStackTrace();  
            }  
        }  
  
        synchronized  
        {  
            public void get()  
            {  
                try  
                {  
                    if (value_in_x == false)  
                    {  
                        wait();  
                    }  
                    else  
                    {  
                        System.out.println("I have taken from " + x);  
                        value_in_x = false;  
                        notify();  
                    }  
                }  
            }  
        }  
    }  
}
```

class Producer extends Thread

{  
Queue a;

public Producer (Queue m)

{  
a=m;

}  
public void run ()

{  
int j=0;  
while (true)

{  
a. peek Cj++);  
}

}

class Consumer extends Thread

{  
Queue b;

public Consumer (Queue n)

{  
b=n;  
}

public void run ()

{  
while (true)

{  
b. get();

{

{

{

{

class Thread

{

    sum C String args [ ]

}

    Queue q = new Queue();

    Producer p = new Producer(q);

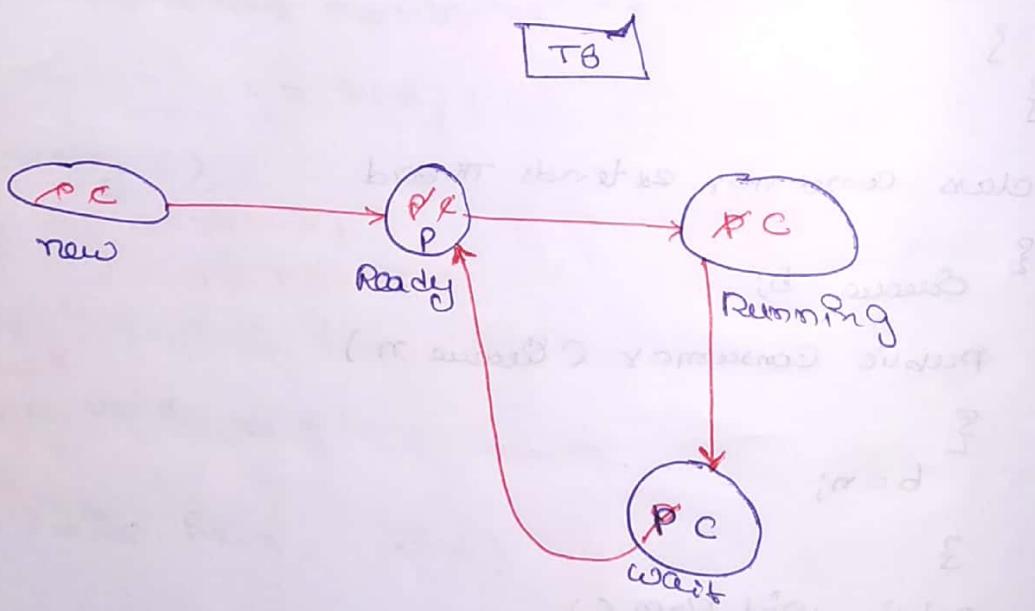
    Consumer c = new Consumer(q);

    p.start();

    c.start();

}

}



- \* wait() and notify() have to be used in synchronized environment in order to avoid "Illegal Monitor State Exception"

## Multi-Threading

It is a process of achieving concurrent executions so that independent activities can be executed independently by creating multiple stacks and threads, for each stack, so that CPU time is efficiently utilised.

### Disadvantage :-

The major disadvantage of multithreading is Synchronisation can't be achieved. But, however this disadvantage can be overcome by using synchronised keyword. Synchronisation can be achieved by using join() also. But, however it is not a right solution becoz the order of execution can't be changed and it is fixed.

### Program

```
class Launch implements Runnable
```

```
{
```

```
    public void run (String args[])
```

```
{
```

```
    ThreadGroup gpl = new ThreadGroup ("Group-1");
```

```
    Launch temp = new Launch();
```

```
    Thread t1 = new Thread (gpl, temp, "A");
```

```
    Thread t2 = new Thread (gpl, temp, "B");
```

`s.o.pln(t1); // Thread [a, 5, Group-1]`

`s.o.pln(t2); // Thread [b, 5, Group-1]`

3

The thread group of a thread is decided during the thread object creation and it can't be changed once the thread is created. Since programmers is not creating main thread changing the group of main thread is not possible.

JVM creates the main thread and the default thread group is called "main". If the programmer doesn't specify the thread groups during thread object creation then all the threads that are created belong to this default thread group main.

## Collections

In order to store data the variable approach was followed. Variable approach suffered from two disadvantages

- i) creation is difficult
- ii) Accessing the data after storage is difficult.

In order to overcome this disadvantages arrays were used. but however arrays also have few disadvantages.

- i) The size of an array is fixed.
- ii) Only Homogeneous data can be stored.
- iii) Demands contiguous memory locations.

In order to overcome this disadvantages of arrays in java 1.2 "collections" were introduced by a computer scientist called Joshua. Collections framework is a

combination of several built-in classes. They are

- i) ArrayList (Dynamic Array)
- ii) LinkedList (Doubly Linked List)
- iii) ArrayDeque (Double Ended Queue)
- iv) PriorityQueue (Min heap)
- v) TreeSet (Balanced Binary Search Tree)
- vi) HashSet [Hashing algorithm (Hash function & Hash table)]
- vii) LinkedHashSet [Hashing algorithm (Hash function & Hash table)]

## ArrayList

## Collection

ArrayList internally makes use of dynamic Array data structure. ArrayList can store heterogeneous data and its size can dynamically grow. But, however ArrayList also demands contiguous memory allocation. Since the data structure used is dynamic array.

### Program

```
public class Launch
```

```
{
```

```
    public static void main(String args[])
```

```
{
```

```
    ArrayList al = new ArrayList();
```

```
    al.add(10);
```

```
    al.add(30.5);
```

```
    al.add("Hari");
```

```
    al.add(true);
```

```
    al.add('S');
```

```
    System.out.println(al);
```

```
}
```

### Output

```
[10, 30.5, Hari, true, S]
```

## few inbuilt methods of ArrayList class

```
package p1;  
import java.util.ArrayList;  
public class launch  
{  
    public static void main(String args[])  
    {  
        ArrayList al = new ArrayList();  
        al.add(30);  
        al.add(20);  
        al.add(50);  
        al.add(70);  
        System.out.println(al);  
  
        ArrayList al2 = new ArrayList();  
        al2.add(300);  
        al2.add(200);  
        al2.add(500);  
        al2.add(100);  
        al2.add(700);  
        System.out.println(al2);  
  
        al.addAll(al2);  
        System.out.println(al);  
  
        al.add(5, 99);  
        System.out.println(al);  
  
        ArrayList al3 = new ArrayList();
```

```
a13.add(3000);  
a13.add(2000);  
a13.add(4000);  
a13.add(1000);  
a13.add(7000);  
a12.addAll(a13);  
S.o.println(a12);  
a1.addAll(a13);  
S.o.println(a1.containsAll(a13)); // true  
S.o.println(a13.isEmpty()); // false  
a13.clear();  
S.o.println(a13.isEmpty()); // true  
S.o.println(a1.contains(99)); // true  
S.o.println(a1.contains(9)); // false  
a1.addAll(a13);  
ArrayList a14 = (ArrayList)a12.clone();  
S.o.println(a14);
```

{}

Note :- Using Linked List all the three disadvantages of

Arrays can be overcome because Linked List can store

Heterogeneous data, its size can dynamically increase

and most importantly it utilises dispersed memory locations  
and doesn't demand contiguous memory allocations.

class Solution

```
{ public void printList() {
```

```
}
```

```
    LinkedList<Integer> ll = new LinkedList<Integer>;
```

```
    ll.add(10);
```

```
    ll.add(20);
```

```
    ll.add(30);
```

```
    ll.add(40);
```

```
    ll.add(50);
```

```
    System.out.println(ll) // [10, 20, 30, 40, 50]
```

```
}
```

