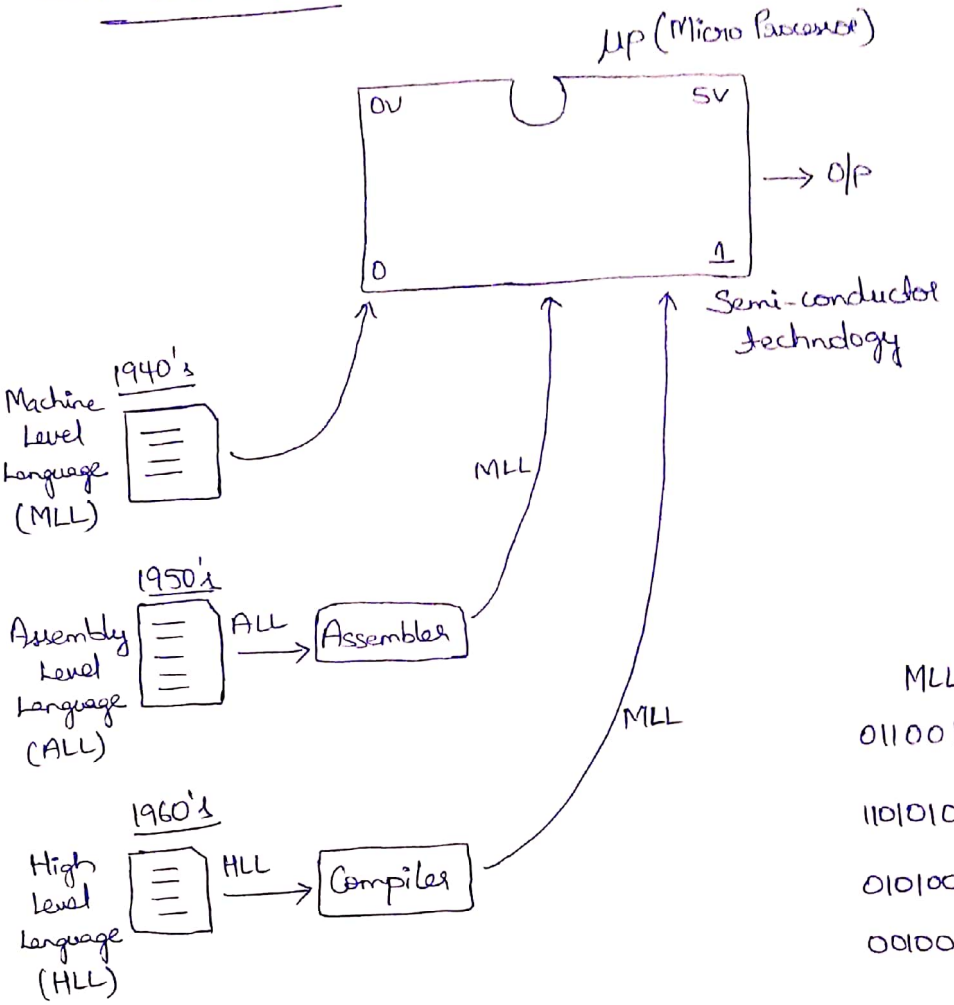
10th/6/19

# JAVA

## INTRODUCTION




| MLL | ALL | HLL |
|---|---|---|
| 0110011 | ADD | + |
| 11010100 | SUB | - |
| 01010011 | DIV | / |
| 00100101 | MUL | * |

NOTE:-

→ Compiler is a software which takes (or) accepts HLL code as input and converts into MLL code as output.

→ Assembler is a software which takes (or) accepts ALL code as input and converts into MLL code as output.

11/6/19

1962

Martin Richards

BCPL

(i) Unstructured

(ii) More memory

1965

Ken Thompson

B

(i) Unstructured

(ii) Less memory

1972

Dennis Ritchie

C

(i) Structured

(ii) Less memory

1982

Stroustroop

C++

(i) Object oriented

(ii) Less memory

(iii) Non-portable

1985

Sun Micro Systems

11 members → Green Team

↓

James Gosling

1986

Russia attacked Afghanistan

1992

Unofficially released

1995

Officially released

James Gosling

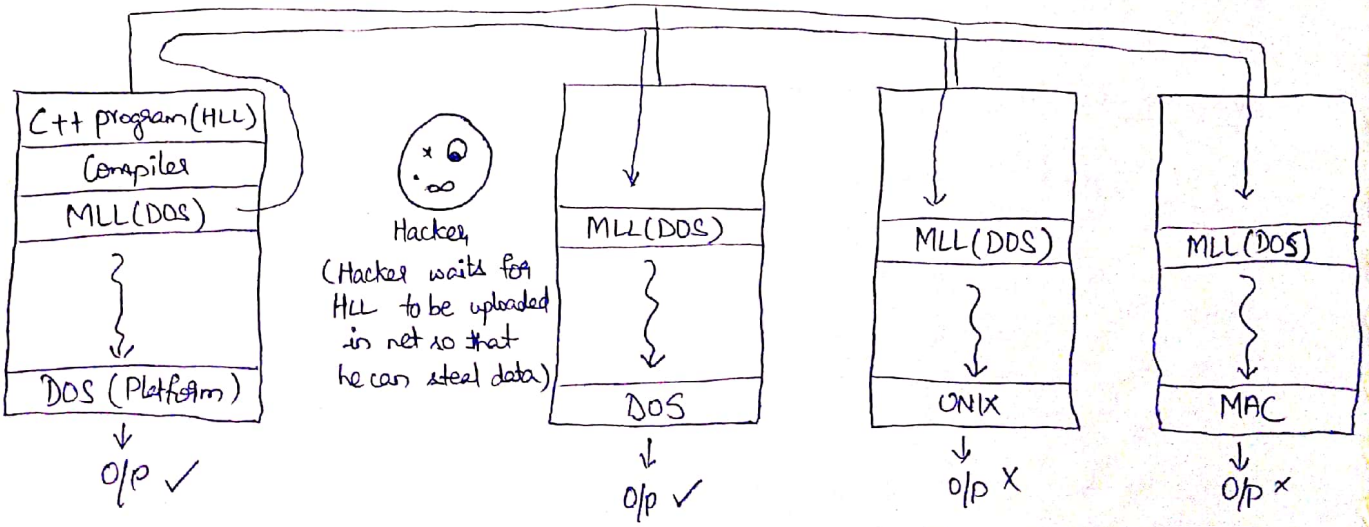
JAVA

(i) Object oriented programming

(ii) Less memory

(iii) Portable

Concept of non-portability :



NOTE

→ Non-portable programming language is a programming language which will work if it is compiled and executed on same OS. If there is a mismatch in platform of compilation and execution, it will not produce any output.

→ Portable programming language is a programming language which will work even if there is a mismatch in platform of compilation and execution.

12/6/19

1914 - World War-I (Began)

1918 — World War-I (End)

1939 — World War-II (Began)

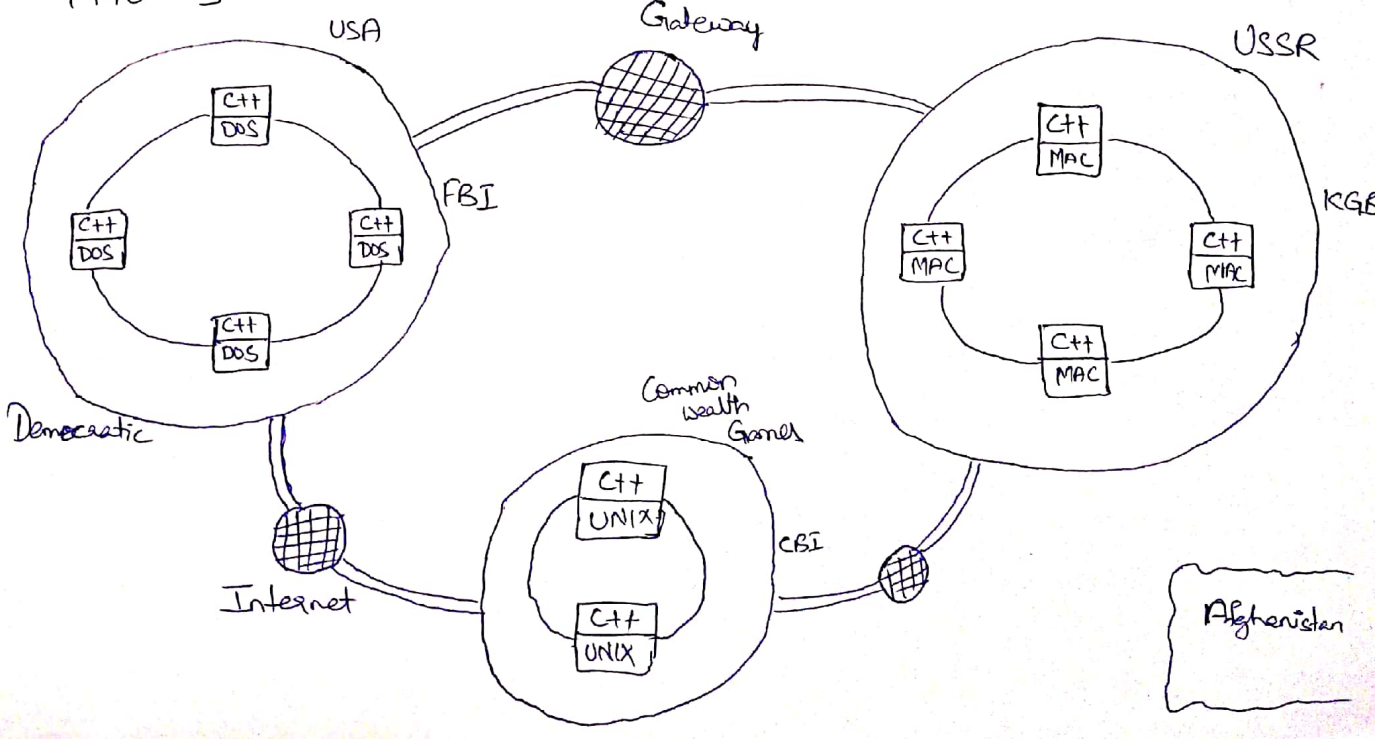
6th August 1945 : Hiroshima Bombed

9th August 1945 : Nagasaki Bombed

1945 — World War-II (Ended)

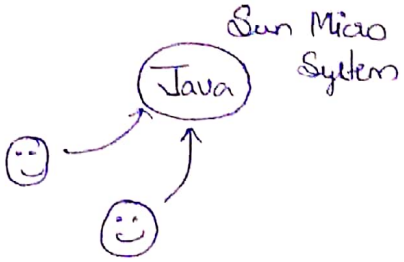
1945 — UNO formed

1950 —
1960 — } Peaceful Decades
1970 —

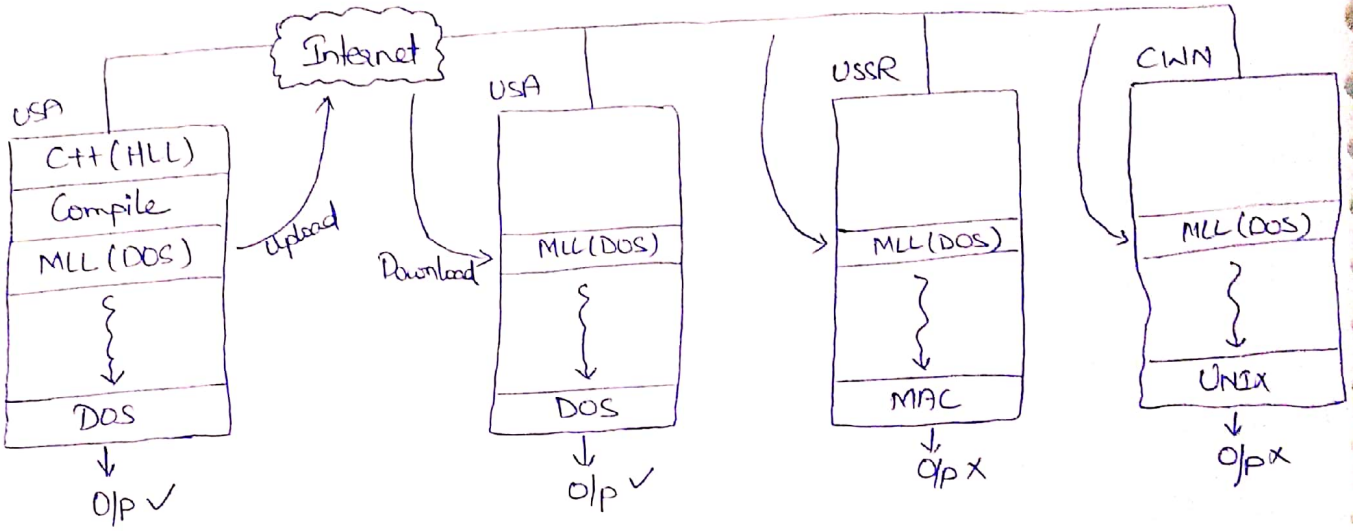


## Main features of Java:-

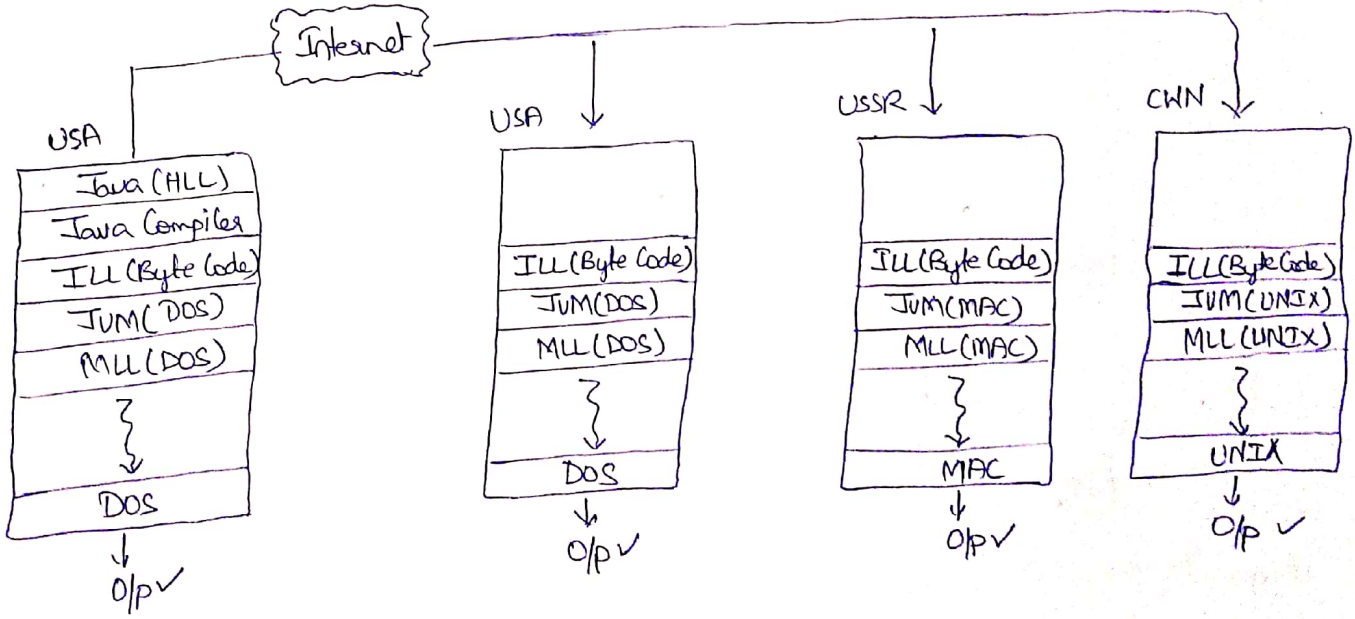
(1) Portable
(2) Open Source
(3) Freely Downloadable




## Failure of C++ and Success of Java:-




## Sucess of java:-



13/6/19

For C++

| Sl.NO | Compilation | Execution | Result |
|---|---|---|---|
| 1 | DOS | DOS | ✓ |
| 2 | DOS | MAC | X |
| 3 | DOS | UNIX | X |

For Java

| Sl.No | Compilation | Execution | Result |
|---|---|---|---|
| 1 | DOS | DOS | ✓ |
| 2 | DOS | MAC | ✓ |
| 3 | DOS | UNIX | ✓ |

| | Platform Dependent | Platform Independent |
|---|---|---|
| HLL | | ✓ |
| ILL | | ✓ |
| MLL | ✓ | |
| JVM | ✓ | |

NOTE:

→ JVM is a software which converts ILL code to MLL code.

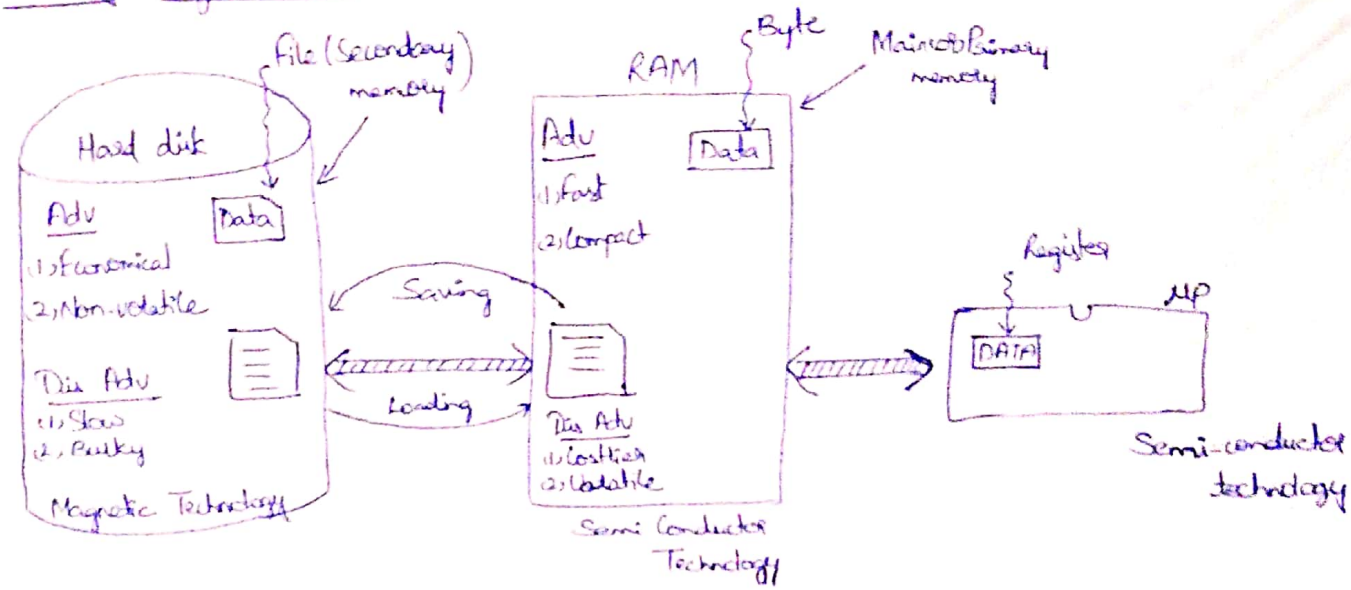
→ JVM is a platform dependent because it is coded using C.

→ HLL code will be present within a file, we call it as source file.

→ ILL code will be present within a file called class file.

→ MLL code will be present within a file called object file.

# Memory Organisation:-




→ There were 4 expectations by computer users

(1) Economical

(2) Non-volatile

(3) Fast

(4) Compact

→ There is no single device which can satisfy all 4 expectations, hence hard disk and RAM both are present

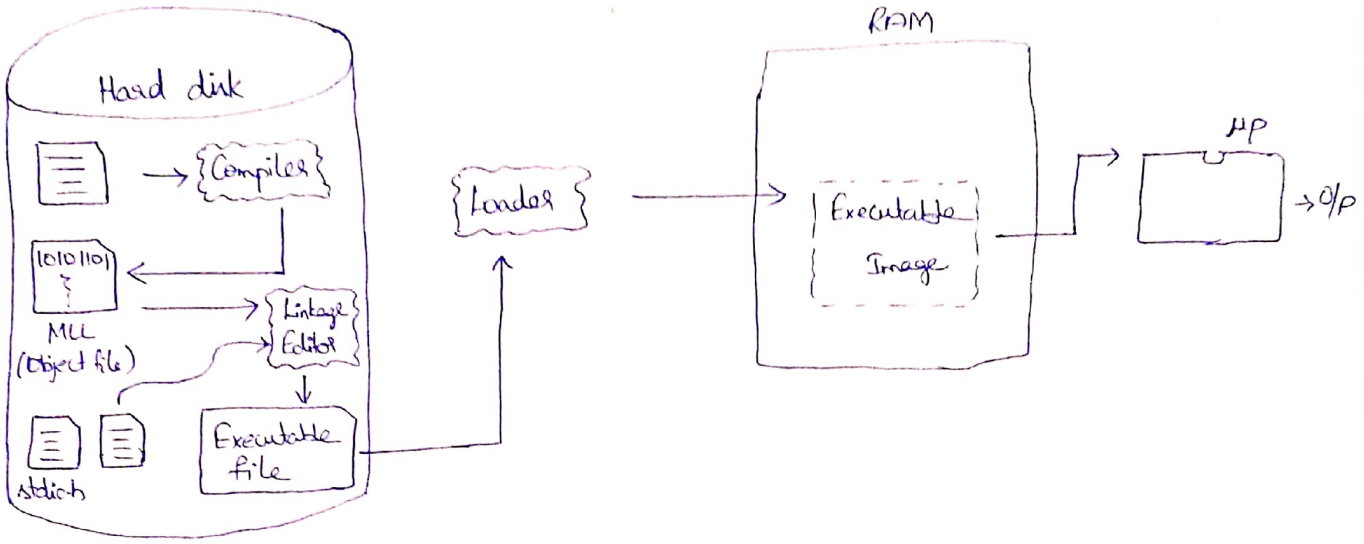
Economical and Non-volatile are satisfied by hard disk

Fast and compact are satisfied by RAM.

→ The copy of file is taken from RAM and stored in hard disk permanently. This process is known as saving.

→ The copy of file is taken from hard disk to RAM for further execution. This process is known as loading.

## Object file v/s executable file:




→ Executable file is a file which contains code in MLL, as it is complete file it can be executable.

→ Object file is a file which contains in MLL, it is a incomplete file and hence it cannot be executed.

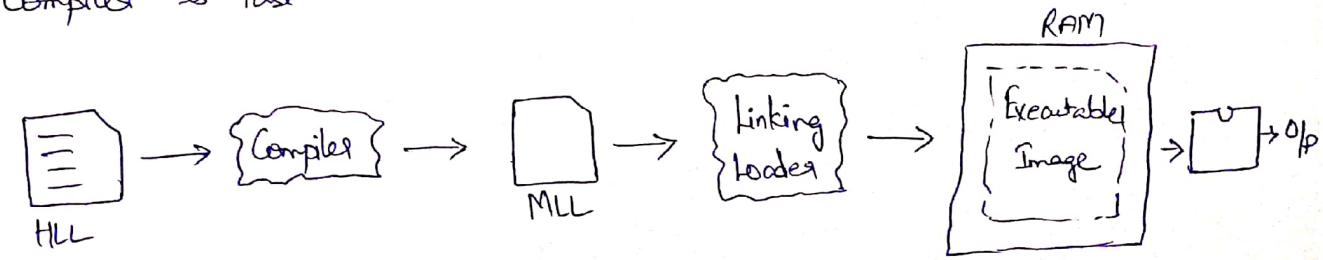
14/6/19

→ Linker is a software which is going to compile MLL and library files as a output, it will produce executable file.

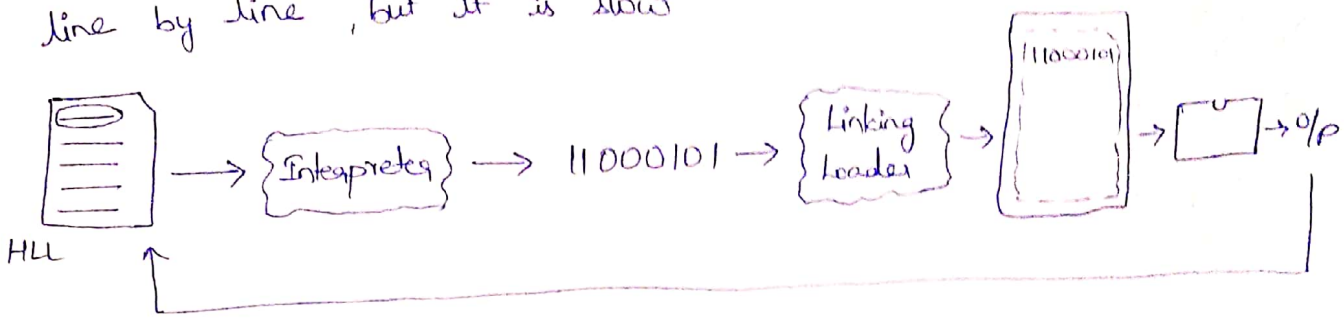
→ Loader is a software which loads executable file from hard disk to RAM.

## COMPILER v/s Interpreter:

→ Compiler is a software which converts HLL to MLL at a time. Compiler is fast.



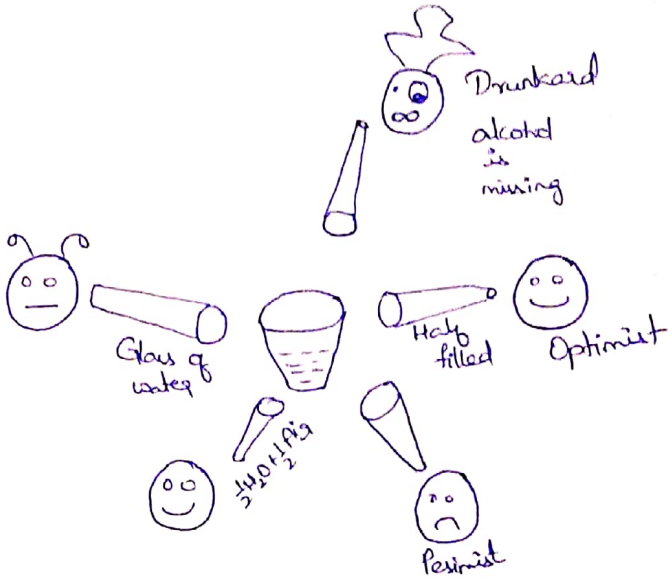
→ Interpreter is a software / translator which converts HLL to MLL line by line, but it is slow

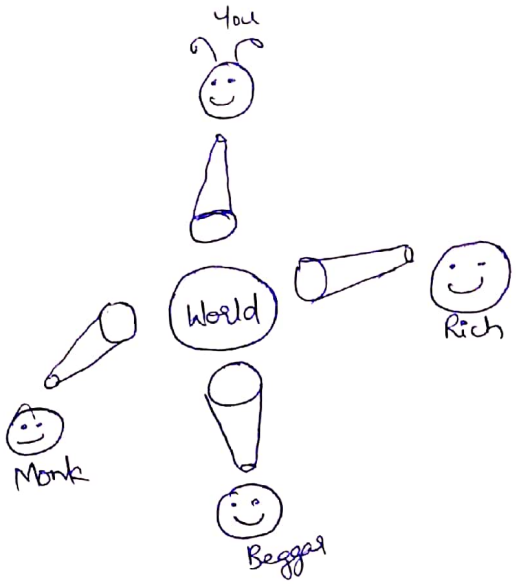


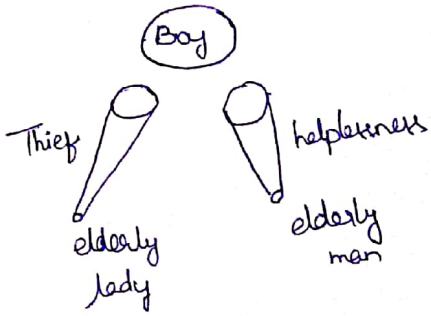

## Object Orientation:-

Orientation:-

(1) Way of looking
(2) Perspective
(3) Point of view

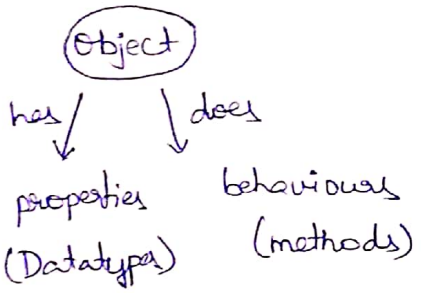


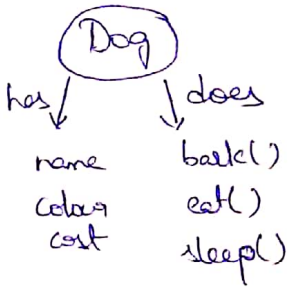



## Principles of Object orientation:

1) Whole world is an collection of objects.
2) No object is useless every object is useful.
3) No object is at isolation, there will be content interaction between objects.
4) Every object belongs to one type, that type is technically called as class (class doesnot exist in reality, objects exist in reality)
5) Every object has something and it does something.
   (has is nothing but properties, does part is nothing but behaviours)




Ex:




### Real time

1) Customer
2) Theatre
3) Movie
4) Queue, Black Ticket Agent
5) Money
6) Ticket
7) Seats

### BMS

1) 


```
class Customer
{
=
}
```

2) 


```
class Movie
{
=
}
```

3) 


```
class Theatre
{
=
}
```

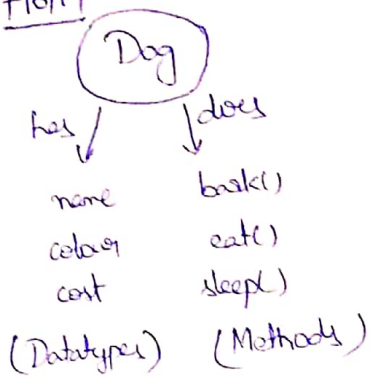
4) 


```
class Seat
{
=
}
```

5) 

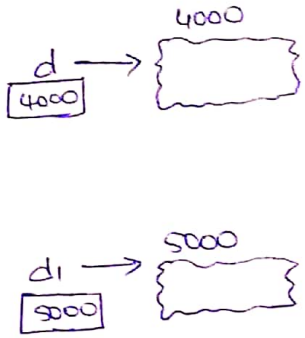

6) 

17/6/19



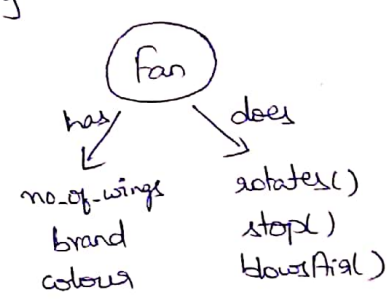

```
class Dog
{
  String name;
  String colour;
  float cost;

  void bark()
  {
  }
  void eat()
  {
  }
  void sleep()
  {
  }
}
```

```
Dog d=new Dog();
d.sleep();
d.bark();
d.eat();
Dog d1=new Dog();
d1.sleep();
```



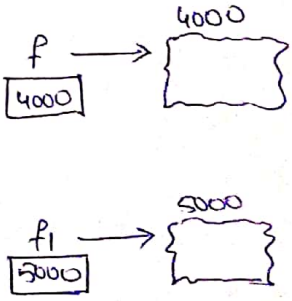

house → contractor → house object
(Blueprint) (Mobile number)

Dog → JVM → Dog object
(Blueprint) (new keyword)




```
class Fan
{
  int no_of_wings;
  String brand;
  String colour;
  void rotates()
  {
  }
  void stop()
  {
  }
  void blowsAir()
  {
  }
}
```

```
Fan f=new Fan();
f.rotates();
f.stop();
f.blowsAir();

Fan f1=new Fan();
f.rotates();
f.stop();
```



## Camel Convention:

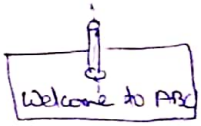
→ classname should begin with capital letter, if there are multiple words in classname all words first letter should be capital.

→ variable names should be completely written in small letters.

→ method names should begin with small letter, if there are multiple words in method, remaining words first letter should be capital.

| OS | PL |
|---|---|
| Windows (Microsoft) | C (Turbo C) |
| MAC (Apple) | C++ (Turbo C) |
| UNIX (AT&T Bell Labs) | Java (Oracle) |

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




```
output name(input)
{
  == // Activity
}
```

```
return type  name (Parameters)
{
  == // Body of method
}
```

→ Main method should be public because it should be visible for operating system.

→ Main method should be static so that OS can access main method without even creating the object.

→ If main method is not public, you will get main method not public error.

→ If main method is not static, you will get main method not static error

→ In the signature of main method the position of public and static can be interchanged.

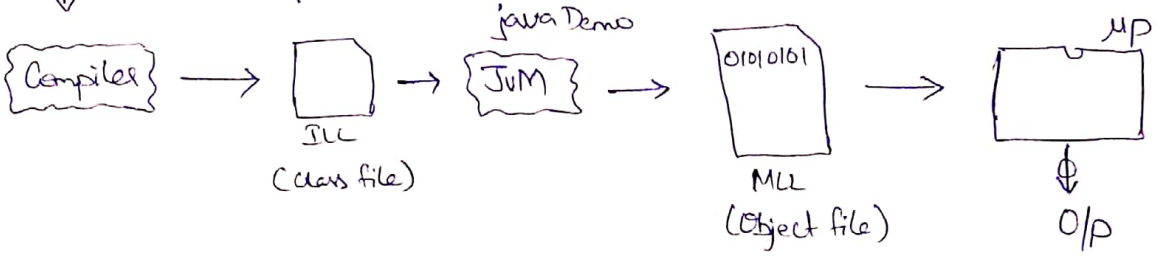
13/6/19

Demo.java

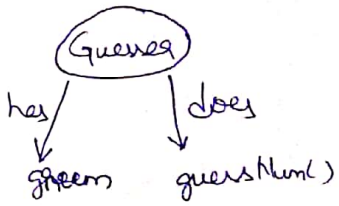
```
class Demo
{
  public static void main(String args[])
  {
    System.out.println(" Welcome to ABC");
  }
}
```

HLL (Source file)

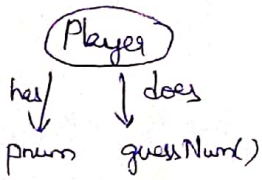
javac Demo.java




## Guesser Game

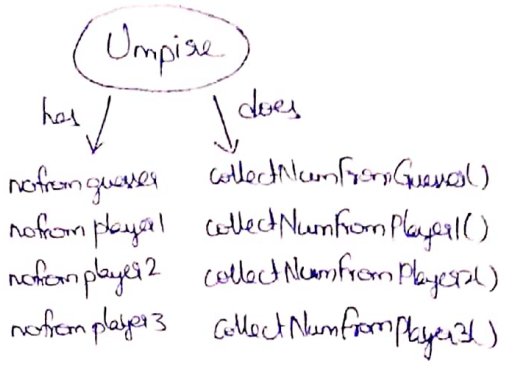


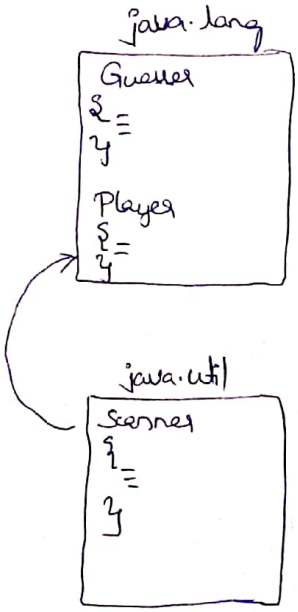

```
import java.util.Scanner;
class Guesser
{
  int gnum;
  int guessNum()
  {
    System.out.println("guesser kindly guess one number");
    Scanner scan=new Scanner(System.in);
      gnum=scan.nextInt();
      return gnum;
  }
}
```




```
class Player
{
    int pnum;
    int guessNum()
    {
      System.out.println(" Player kindly guess one number");
        Scanner scan=new Scanner(System.in)
                pnum=scan.nextInt();
                return pnum;
    }
```

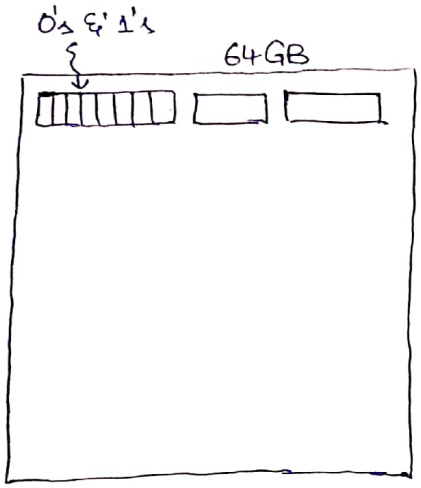
```
class Umpire
{
   int nofromguesser1;
   int nofromplayer1;
   int nofromplayer2;
   int nofromplayer3;

   void collectNumfromGuesser()
   {
      Guesser g=new Guesser();
      nofromguesser=g.guessNum();
   }
   void collectNumfromPlayer()
   {
      Player p1=new Player();
      Player p2=new Player();
      Player p3=new Player();
      nofromplayer1=p1.guessNum();
      nofromplayer2=p2.guessNum();
      nofromplayer3=p3.guessNum();
   }
   void compare()
   {
      if(nofromguesser==nofromplayer1)
         System.out.println("Player1 wins");
      else if(nofromguesser==nofromplayer2)
         System.out.println("Player2 wins");
      else if(nofromguesser==nofromplayer3)
         System.out.println("Player3 wins");
      else
         System.out.println("Gameover...!! Try Again");
   }
   class LaunchUmpire
   {public static void main(String args[])
      {
         Umpire u=new Umpire();
         u.collectNumfromGuesser();
         u.collectNumFromPlayer1();
         u.compare();
      }
   }
```



19th 6/19

| Real life | Java Data types | format |
|---|---|---|
| Character | char | UTF |
| Integer | byte, short, int, long | base-2 |
| Real numbers | float, double | IEEE |
| Yes/No or, True/false | boolean | JVM Dependent |
| Photo | Inbuilt classes | JPEG |
| Audio | Inbuilt classes | mp3 |
| Video | Inbuilt classes | mp4 |




$64 \times 10^9$ B

64 x 1000000000 B

64000000000 x 8 bits

64000000000 x 8 x 2

→ Format is a technique or a software which will take java data types and it will convert it into 0's and 1's.

Character type Real world data :-

→ To manage character type real world data java has provided 'char' data type.

→ In 'C' programming language 'char' data type is of 1 byte where as in java character data type is of 2 bytes because in 'C' programming language there were 128 symbols. For representing 128 symbols 7 bits are sufficient. In java there are 65,536 symbols and for representing them 16 bits are required nothing but 2 bytes

A - 0
B - 0

2 symbol

$2^1$ ⇒ 1 byte

A - 00
B - 01
C - 10
D - 11

4 symbols

$2^2$ ⇒ 2 bits

A - 000
B - 001
C - 010
D - 011
E - 100
F - 101
G - 110
H - 111

8 symbols

$2^3$ ⇒ 3 bits

128 (ASCII) (1 Byte)

1st sym = 0000000
2nd sym = 0000001
3rd sym = 0000010
⋮
127th sym = 1111110
128th sym = 0111111

128 symbols

$2^7$ ⇒ 7 bits

65536 (UTF) (2 Bytes)

1st sym = 0000 0000 0000 0000
2nd sym = 0000 0000 0000 0000
⋮
65535th sym = 1111 1111 1111 1110
65536th sym = 1111 1111 1111 1111

65536 symbols

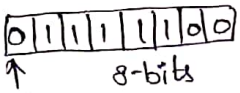
$2^{16}$ ⇒ 16 bits

## integer type real world data :-

byte a = 124

2 | 124
2 | 62 — 0
2 | 31 — 0
2 | 15 — 1
2 | 7 — 1
2 | 3 — 1
1 — 1

1111100




### (i) Byte [1 byte = 8 bits]

Ex: Age of a person

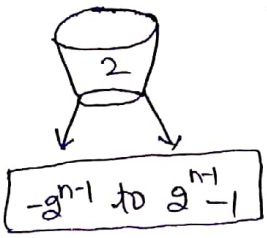

$-2^{8-1}$ to $2^{8-1}-1$

$-2^7$ to $2^7-1$

−128 to 127

### (ii) short [2 byte = 16 bits]

Ex: Salary of a fresher



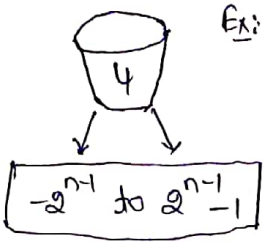

$-2^{16-1}$ to $2^{16-1}-1$

$-2^{15}$ to $2^{15}-1$

−32768 to 32767

### (iii) int [4 byte = 32 bits]

Ex: Population of country




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

$-2^{31}$ to $2^{31}-1$

−2147483648 to 2147483647

### (iv) long [8 bytes = 64 bits]

Ex: Population of whole world


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

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

−9223372036 854775808 to 9223372036 854775807

→ In real world, this integer datatype is not fixed. It is of varying magnitude. In java also, there are 4 datatypes

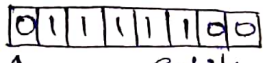
(i) Byte
(ii) Short
(iii) int
(iv) long

20/6/19

byte a=124

2|124
2|62 — 0
2|31 — 0
2|15 — 1
2|7 — 1
2|3 — 1
2|1 — 1

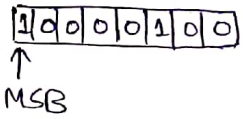
1111100




byte a=-124

124 = 01111100

124 = 10000011 → 1's complement
+1
-124 = 10000100 2's complement



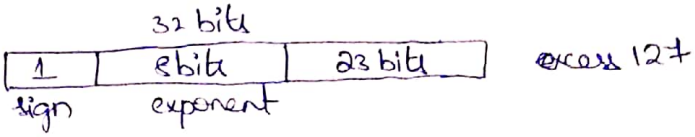

NOTE:-

(1) Negative numbers in integers will be stored in 2's complement
(2) MSB bit for +ve numbers in its binary format is '0' where as MSB bit for -ve numbers in its binary format is '1'.
(3) 0 is considered as +ve numbers because MSB bit of '0' in its binary format is '0'.

## Real number type real world data:

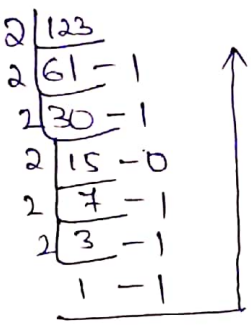
To manage real number type real world data java had provided float and double data types.

1) float





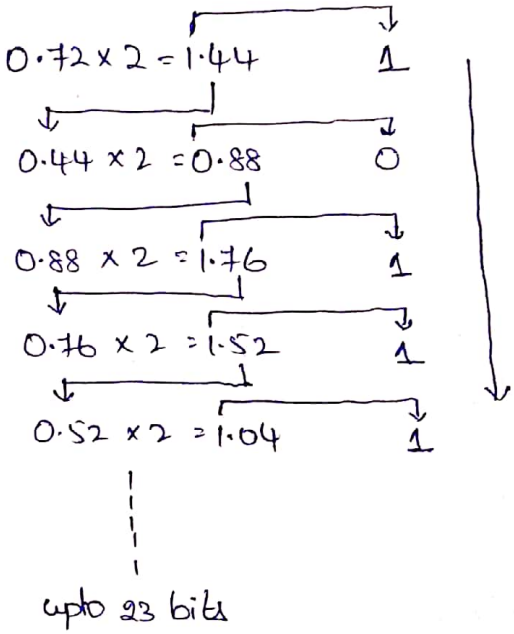
float a=123.72




123 = 1111011

0.72 = 10111---

123.72 = 1111011.10111---




upto 23 bits

No of positions

According to IEEE engineers, decimal point should be shifted after the first bit.

1111011.10111---

1.1110111011---

mantissa

No. of positions shifted is n

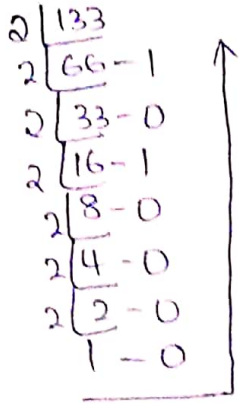
Multiply it with $2^n$

$1.1110111011 \times 2^6$

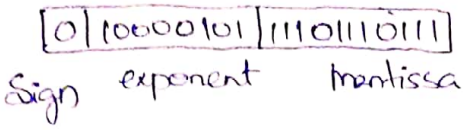
mantissa

excess is 127 as standard

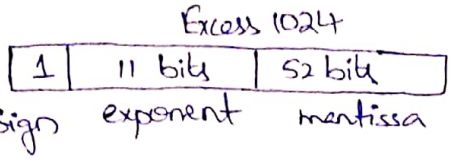
n+127 = 6+127 = 133




10000101
exponent




12; double


Here excess 1024

In IEEE format floating point number will follow something called as single precision format.

In IEEE format double valued numbers will be following double precision format.

```
byte a=124;
double b;
   b=a;
System.out.println(a);
System.out.println(b);
```

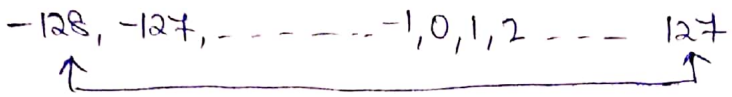

O/p:- 124
124.0

```
double a=123.72;
  byte b;
   b=a;
System.out.println(a);
System.out.println(b);
```


O/p:- error

```
double a=124.72;
byte b;
b=(byte)a;
System.out.println(a);
System.out.println(b);
```

O/p:- a=124.72
124




Typecasting:-

It is a process of converting one type of data another type of data.

Yes/No type real world data:-

To manage Yes/NO type real world data java has provided boolean data type. The no. of bytes allocated for boolean datatype is JVM dependent.

21/6/19

## INCREMENTS | DECREMENTS

(1) int a=5;

int b= a++ + ++a + ++a + ++a + a++;
5 7 8 9 9

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

O/P: 10
38

a 5 6 7 8 9 10
b

(2) int a=5;

int b= ++a + ++a + a++ + a++ + a++ + ++a + a++;
6 7 7 8 9 11 11

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

O/P: 12
59

a 5 6 7 8 9 10 11 12
b

(3) int a=5;

int b= ++a + a++ + --a + --a + ++a - a-- + ++a + a++ - ++a - --a;
6 + 6 6 5 6 6 6 6 8 7

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

O/P: 7
10

a 5 6 5 6 5 6 5 6 7
b

**Pre-increment/decrement : (++a, --a)**

(1) First go to memory
(2) Incre/Decre the value
(3) Consider updated value.

**Post-increment/decrement (a++, a--)**

(1) Consider the present value
(2) Go to memory and increment/decrement the value.

## Pattern Programming:-

```
for (intialize ; condition ; incre/decre)
```

(1) *

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

(2)
*
*
*
*
*

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

(3) *****

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

(4)
*****
*****
*****

i = rows
j = columns

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

1) println("*");

O/p: *

2) print("*");

O/p: *

(5)
```
*
* *
* * *
* * * *
* * * * *
```

i = rows
j = stars

| rows(i) | stars(j) |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |

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

(6)
```
    j=5 4 3 2 1
i=1  * * * * *
2    * * * *
3    * * *
4    * *
5    *
```

i = rows
j = stars

| rows(i) | stars(j) |
|---|---|
| 1 | 5 |
| 2 | 4 |
| 3 | 3 |
| 4 | 2 |
| 5 | 1 |

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

(7)
```
-----* * * * *
------ * * * *
------ * * * *
------ * * * *
------ * * * *
```

i = rows
j = spaces
k = stars

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

(8)

```
        j
    1 2 3 4 5 6
i=1  * * * * *
2    * * * * *
3      * * * * *
4        * * * * *
5          * * * * *
```

i = rows
j = spaces
k = stars

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

(9)

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

i = rows
j = spaces
k = stars

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

(10)

```
    j:
    5 4 3 2 1
i:
1 - - - - - *
2 - - - - * *
3 - - - * * *
4 - - * * * *
5 - * * * * *
```

i = rows
j = spaces
k = stars

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

(11)

```
    5 4 3 2 1
1 - * * * * *
2 - - * * * *
3 - - - * * *
4 - - - - * *
5 - - - - - *
```

i = rows
j = spaces
k = stars

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

(12)

```
- - - - - *
- - - - * * *
- - - * * * * *
- - * * * * * * *
- * * * * * * * * *
```

i = rows
j = spaces
k = stars

| rows(i) | spaces(j) | stars(k) |
|---|---|---|
| 1 | 5 | 1 = 2x1-1 |
| 2 | 4 | 3 = 2x2-1 |
| 3 | 3 | 5 = 2x3-1 |
| 4 | 2 | 7 = 2x4-1 |
| 5 | 1 | 9 = 2x5-1 |

```
for(int i=5; i<=5; i++)
{
 for( int j=5; j>=i; j--)
  {
   System.out.print(" ");
  }
 for(int k=1; k<=(2xi)-1; k++)
  {
   System.out.print(" ");
  }
 System.out.println(" ");
}
```

(13)

```
- * * * * * * * * *
- - * * * * * * *
- - - * * * * *
- - - - * * *
- - - - - *
```

i = rows
j = spaces
k = stars

| i | j | k | |
|---|---|---|---|
| 1 | 1 | 9 | 11-2x1 |
| 2 | 2 | 7 | 11-2x2 |
| 3 | 3 | 5 | 11-2x3 |
| 4 | 4 | 3 | 11-2x4 |
| 5 | 5 | 1 | 11-2x5 |

```
for(int i=1; i<=5; i++)
{
 for(int j=1; j<=i; j++)
  {
   System.out.print(" ");
  }
 for(int k=11-(2*i); k>=1; k--) / for(int k=1; k<=11-(2*i); k++)
 {
  System.out.print("*");
 }
 System.out.println(" ");
}
```

(14)

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

i: rows
j: stars

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

(15)

```
for(int i=1; i<=5; i++)
{
  for(int j=5; j>=i; j--)
  {
    System.out.print(" ");
  }
  for(int k=(2*i)-1; k<=i; k++)
  {
    System.out.print("*");
  }
  System.out.println();
}
for(i=6; i<=10; i++)
{
  for(j=1; j<=1; j++)
  {
    System.out.print(" ");
  }
  for(k=10; k<=15; k++)
  {
    System.out.print("*");
  }
  System.out.println();
}
```

(15)
```
for(int i=1; i<=5; i++)
{
  for(int j=5; j>=i; j--)
  {
    System.out.println(" ");
  }
  for(int k=(2xi)-1; k<=i; k++)
  {
    System.out.println("*");
  }
 System.out.println();
}
for(int i=5; i>=1; i++)
{
  for(int j=1; j<=i; j++)
  {
    System.out.print(" ");
  }
  for(int k=(2*i)-1; k>=1; k--)
  {
    System.out.print("*");
  }
 System.out.println();
 }
}
}
```

24/6/19

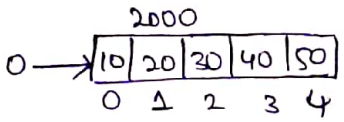
## Arrays:-

Traditional / Conventional approach:-

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

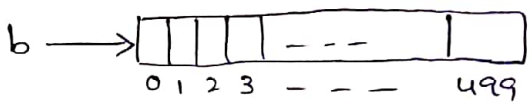
Array approach:-

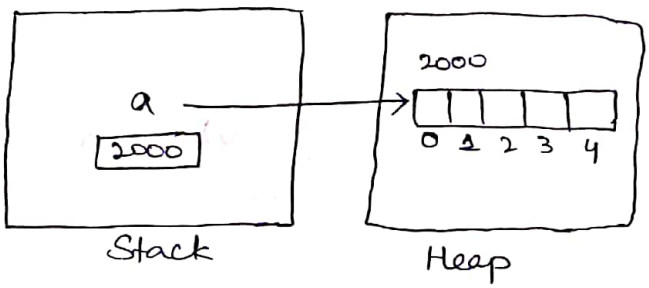
int a[] = new int[5];




int b[] = new int[500]

b[0] = 100;
b[2] = 200;
b[396] = 300;




Different types of array datastructures:-

1) One dimensional array (1-D)
2) Two dimensional array (2-D)
3) Three dimensional array (3-D)

(1) One dimensional array (1-D):-

```
import java.util.Scanner;
class Demo
{
  public static void main(String args[])
  {
    int a[]=new int[5];
    Scanner scan=new Scanner(System.in);
    for(int i=0; i<=a.length-1; i++)
    {
      System.out.println("Enter the marks");
      a[i]=scan.nextInt();
    }
    System.out.println("Entered marks are");
    for(int i=0; i<=a.length-1; i++)
    {
      System.out.println(a[i]);
    }
  }
}
```

5 students

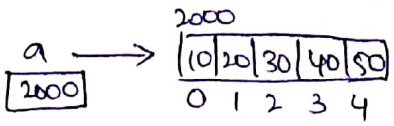
0 → 1st student

1 → 2nd

2 → 3rd

3 → 4th

4 → 5th




(2) Two dimensional array (2-D):-

```
import java.util.Scanner;
class Demo1
{
  public static void main(String args[])
  {
    Scanner scan=new Scanner(System.in);
    int a[][]=new int[3][5];
    for(int i=0; i<=a.length-1; i++)
    {
      for(int j=0; j<=a[i].length-1; j++)
      {
        System.out.println("Enter the marks of class"+i+"student"+j);
        a[i][j]=scan.nextInt();
      }
    }
  }
}
```

| class(i) | student(j) |
|---|---|
| class 0 | 5 |
| class 1 | 5 |
| class 2 | 5 |

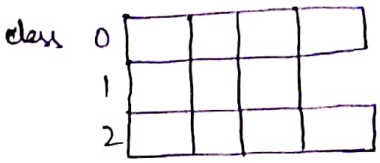
| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| class 0 | 10 | 20 | 30 | 40 | 50 |
| class 1 | | | | | |
| class 2 | | | | | |

```
for (int i=0;i<=a.length-1; i++)
{
  for (int j=0; j<=a[i].length-1; j++)
  {
    System.out.println("Entered marks of class "+i+"students "+j);
    System.out.println(a[i][j]);
  }
}
}
}
```

2-D Jagged Array:-

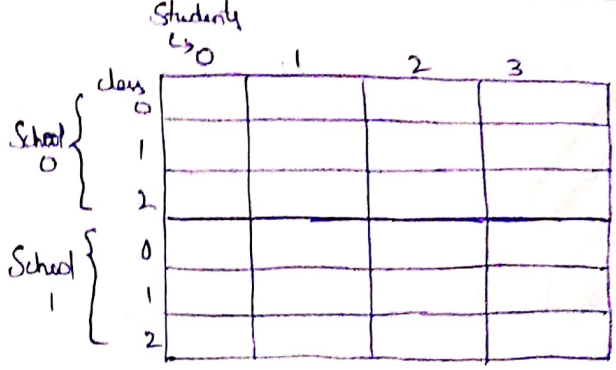
```
import java.util.Scanner;
class Demo2
{
  public static void main(String args[])
  {
    Scanner Scan=new Scanner(System.in)
    int a[][]=new int[3][];
    a[0]=new int[4];
    a[1]=new int[3];
    a[2]=new int[4];

    for(int i=0; i<=a.length-1;i++)
    {
      for(int j=0; j<=a[i].length-1;j++)
      {
        System.out.println("Enters marks of class"+i+"Students "+j);
        a[i][j]= scan.nextInt();
      }}
    for(int i=0;i<=a.length-1;i++)
    {
      for(int j=0; j<=a[i].length-1;j++)
      {
        System.out.println("Entered marks of class"+i+"student "+j);
        System.out.println(a[i][j]);
      }
    }
  }
}
```

| class(i) | student (j) |
|---|---|
| class 0 | 4 |
| class 1 | 3 |
| class 2 | 4 |



## Three dimensional array:-

| School | class | students |
|---|---|---|
| 0 | 0 | 4 |
| | 1 | 4 |
| | 2 | 4 |
| 1 | 0 | 4 |
| | 1 | 4 |
| | 2 | 4 |




```
import java.util.Scanner;
class Demo
{
  public static void main(String args[])
  {
    Scanner scan=new Scanner(System.in);
    int a[][][]=new int[2][3][4];
    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+"students"+k);
          a[i][j][k]=scan.nextInt();
        }
      }
    }
    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("Entered marks of school"+i+"class"+j+"students"+k);
          System.out.println(a[i][j][k]);
        }
      }
    }
  }
}
```

## 3-D Jagged Array:

| School | class | Students |
|---|---|---|
| 0 | 0 | 4 |
| | 1 | 3 |
| | 2 | 4 |
| 1 | 0 | 4 |
| | 1 | 3 |
| | 2 | |

```
import java.util.Scanner;
class Demo4
{
  public static void main(String args[])
  {
    Scanner scan=new Scanner(System.in)
    int a[][][]=new int[2][][];
    a[0]=new int[3][];
    a[1]=new int[2][];
    a[0][0]=new int[4];
    a[0][1]=new int[3];
    a[0][2]=new int[4];
    a[1][0]=new int[3];
    a[1][1]=new int[4];
    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 marks of school"+i+"class"+j+"students"+k);
          a[i][j][k]=scan.nextInt();
        }
      }
    }
    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("Entered marks of school"+i+"class"+j+"students"+k);
          System.out.println(a[i][j][k]);
        }
      }
    }
  }
}
```

25/6/19

## Advantages of Arrays:-

(1) Creation is simple.

(2) Storing and retrieving also simple.

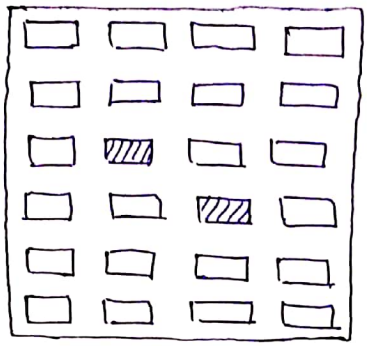
## Disadvantages of Arrays:-

(1) It can accept only homogeneous data.

(2) Once array size is fix it cannot grow or shrink.

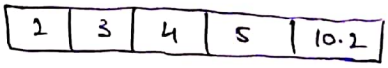
(3) Arrays demand continuous memory locations, it cannot store the data in discrete memory locations.

int a=new int[5];



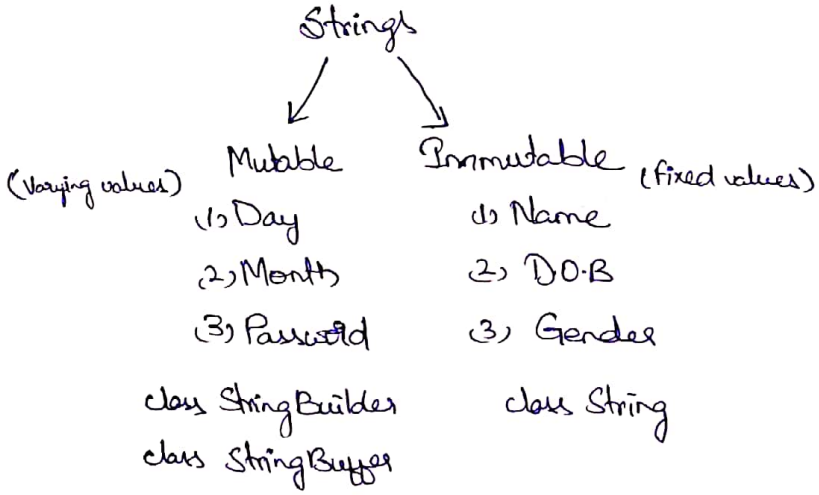
int a[]=new int[5]; ✓

int a[]=new int[5]; ✗



## STRINGS:-

In java, strings are treated as object. In java, strings are not terminated with null character.

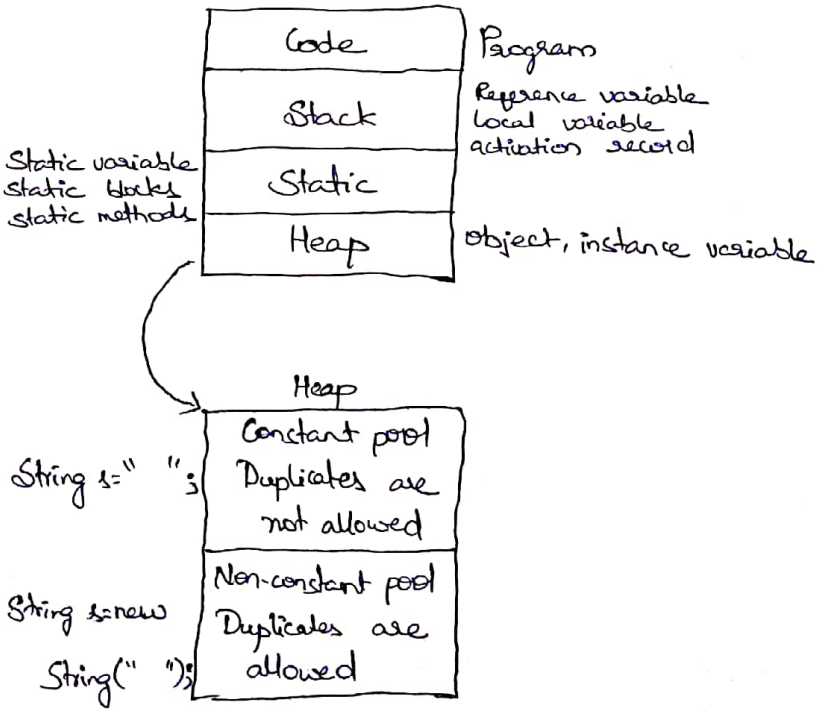



There are three ways of creating immutable strings

1) String s="RAMA";

2) String s=new String("RAMA");

3) char name[] = {'R','A','M','A'}
   String s=new String(name);



For creating immutable strings java has provided "String" class.

For creating mutable strings java has provided two classes "StringBuffer" & "StringBuilder".

There are four ways for comparing the strings

1) equals() → It compares the values of strings

2) == → It compares references (address) of strings.

3) compareTo() → It compares values of strings character by character.

4) equalsIgnoreCase() → It compares the values of strings by ignoring the case.

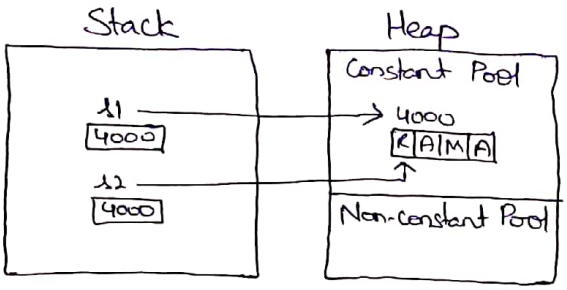
```
1) class Demo1
{
  public static void main(String args[])
  {
    String s1="RAMA";
    String s2="RAMA";
    if(s1.equals(s2))
     System.out.println("Strings are equal");
    else
     System.out.println("Strings are unequal");
  }
}
```

Op: Strings are equal.

(2) class Demo2

```
{
  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");
  }
}
```

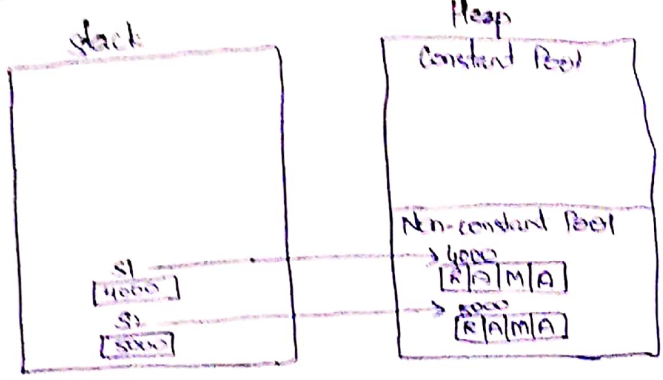
O/P:- References are equal.




(3) class Demo1

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

O/P:- Strings are equal.



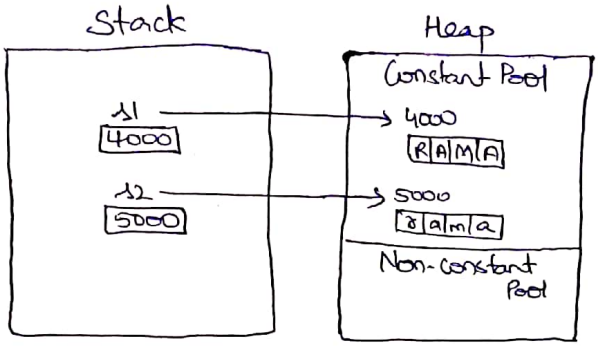

4) class Demo4
```
{
  public static void main(String args[])
  {
    String s1=new String("RAMA");
    String s2=new String("RAMA");
    if(s1==s2)
      System.out.println("References are equal");
    else
      System.out.println("References are unequal");
  }
}
```
O/P:- References are unequal

5) class Demo5
```
{
  public static void main(String args[])
  {
    String s1="RAMA";
    String s2="rama";
    if(s1.equals(s2))
      System.out.println("Strings are equal");
    else
      System.out.println("Strings are unequal");
  }
}
```
O/P:- Strings are unequal.

(6) class Demo2

```
{
  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: References are unequal.




(7) Ignore case

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

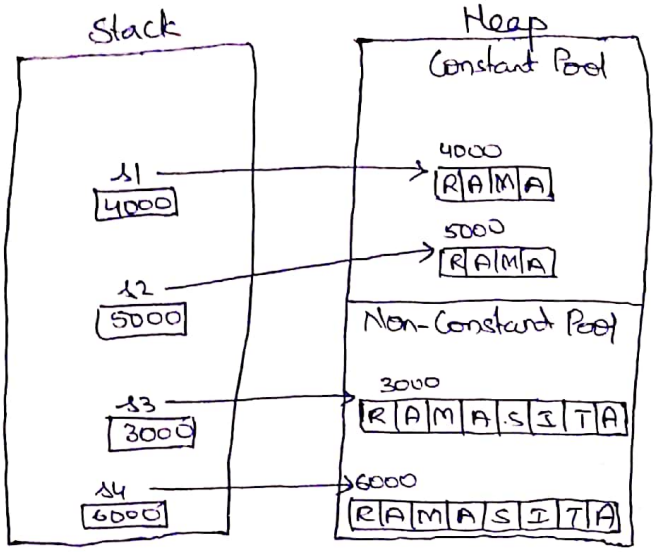
O/P: Strings are equal.

8) class Demo
{
public static void main(String args[])
{
String s1="RAMA";
String s2="SITA";
String s3=s1+s2;
String s4=s1+s2;
if(s3.equals(s4))
System.out.println("Strings are equal");
else
System.out.println("Strings are unequal");
}
}

O/p:- Strings are equal.

9) class Demo
{
public static void main(String args[])
{
String s1="RAMA";
String s2="SITA";
String s3=s1+s2;
String s4=s1+s2;
if(s3==s4)
System.out.println("Refernces are equal");
else
System.out.println("References are unequal");
}
}

O/p:- References are unequal.




(10) String s1="SITA";

System.out.println(s1);

s1.concat("RAVANA");

System.out.println(s1);

O/p:- SITA
SITA

Immutable strings cannot be changed in value so after concatination it gives some value.

(11)

StringBuffer s1=new StringBuffer("SITA");

System.out.println(s1);

s1.append("RAMA");

System.out.println(s1);

O/p:- SITA
SITARAMA

(12) StringBuilder s1=new StringBuilder("SITA");

System.out.println(s1);

s1.append("RAMA");

System.out.println(s1);

O/p:- SITA
SITARAMA

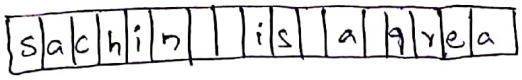
(13) StringBuffer sb=new StringBuffer();

```
System.out.println(sb.capacity());
sb.append("Sachin");
System.out.println(sb.capacity());
sb.append("is a great batsman");
System.out.println(sb.capacity());
sb.append("He is from India");
System.out.println(sb.capacity());
```

O/p:- 16
16
34
70

Intially StringBuffer have 16 bits capacity but while we increase the capacity it vary to 16*2+2 = 34

34*2+2 = 70





| StringBuffer | StringBuilder |
|---|---|
| (1) Intial capacity is 16. | (1) Intial capacity is 16. |
| (2) It is used for creating mutable strings. | (2) It is used for creating mutable strings. |
| (3) Race condition doesnot occur. | (3) Race condition occurs. |
| (4) It is thread safe. | (4) It is not thread safe. |

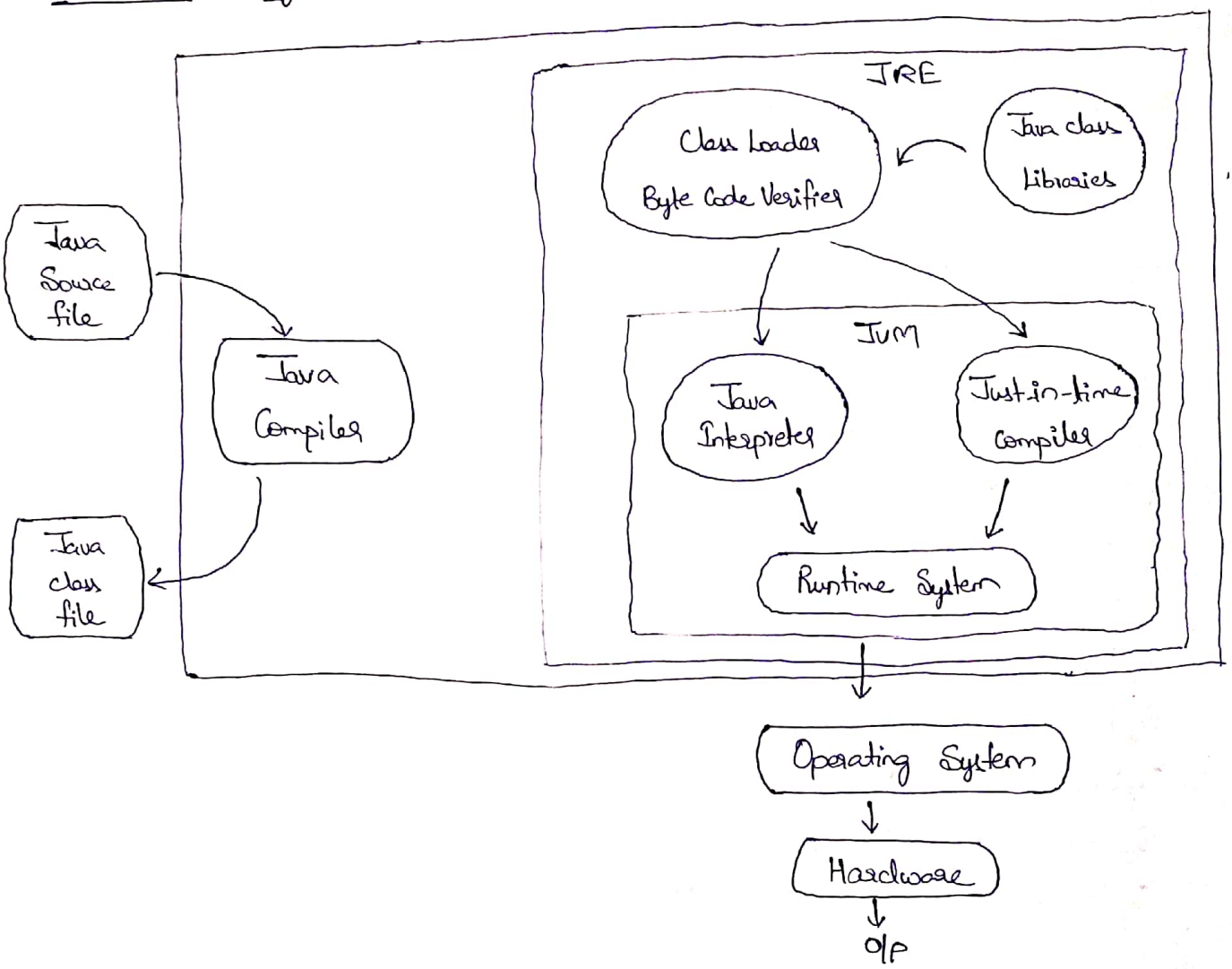
Some of inbuilt methods available on strings:-

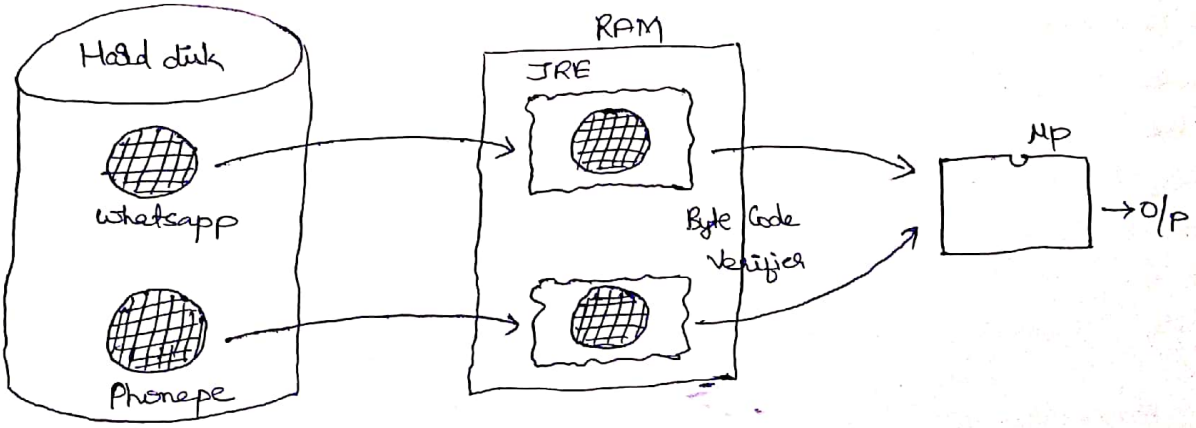
```
class Demo
{
  public static void main(String args[])
  {
    String s1="RajaRamMohanRoy";
    System.out.println(s1.toUpperCase());
    System.out.println(s1.toLowerCase());
    System.out.println(s1.startsWith("Raja"));
    System.out.println(s1.startsWith("Rani"));
    System.out.println(s1.endsWith("Roy"));
    System.out.println(s1.endsWith("Boy"));
    System.out.println(s1.contains("Mohan"));
    System.out.println(s1.contains("Sohan"));
    System.out.println(s1.charAt(3));
    System.out.println(s1.substring(4));
    System.out.println(s1.substring(4,7));
    System.out.println(s1.indexOf('a'));
  }
}
```

O/p:- RAJARAMMOHANROY
rajarammohanroy
true
false
true
false
true
false
a
RamMohanRoy
Ram
1

26/6/19

# Architecture of Java:



# 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 ~~February~~ December 8, 1998

Java 1.3

→ Kestrel

→ Released on May 8, 2000

Java 1.4

→ Merlin

→ Released on February 6, 2002

Java 5

→ Tiger

→ Released on September 30, 2004

Java 6

→ Mustang

→ Released on Dec 11, 2006

Java 7

→ Dolphin

→ Released on July 28, 2011

Java 8

→ Released on 18 March, 2014

Java 9

→ Released on 2017

Java 10

→ Released on March 10, 2018

Java 11

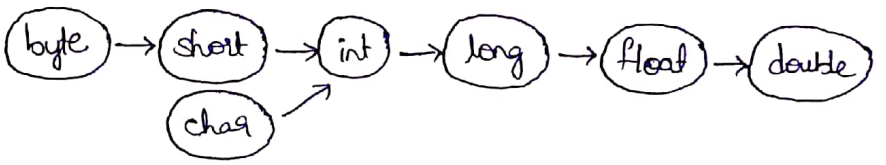
→ Released on September 2018

## Wrapper classes:

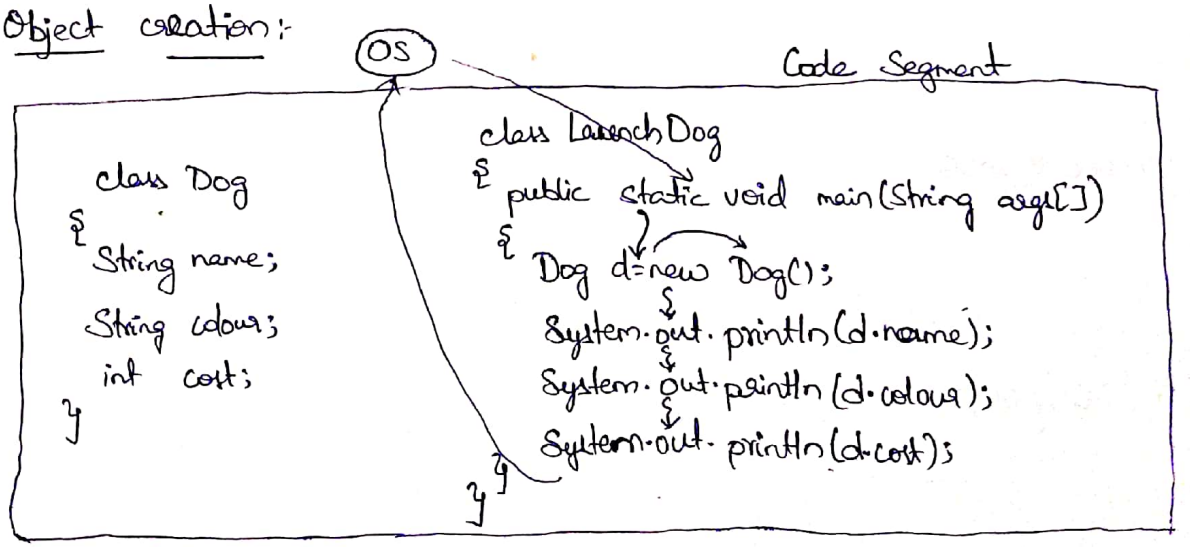
| | | |
|---|---|---|
| int a | Integer | a=new Integer(48); |
| byte b | Byte | b=new Byte(20); |
| short c | Short | c=new Short(30); |
| long d | Long | d=new Long(201010); |
| float e | Float | e=new Float(20.25); |
| double f | Double | f=new Double(202525); |
| char g | Char | g=new Character('a'); |
| boolean h | | |

Java is not 100% pure object oriented programming language but if you want to make project as 100% pure object oriented then we should use wrapper class.

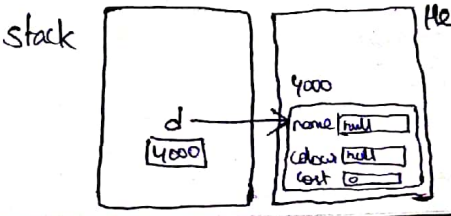
## Numeric promotion chart:




## Object creation:




O/p: null
null
0

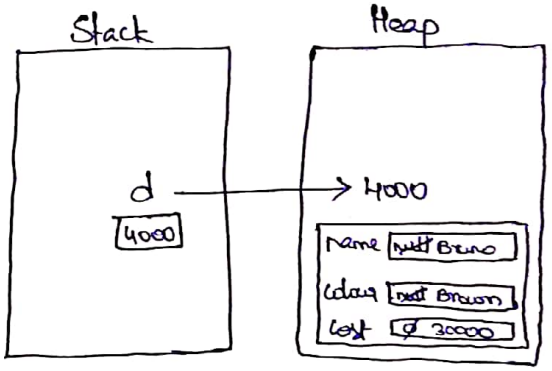


OS

```
class Dog
{
  String name;
  String colour;
    int cost;
}
```

```
class LaunchDog
{
  public static void main(String args[])
  {
    Dog d=new Dog();
    d.name="Bruno";
    d.colour="Brown";
    d.cost=30000;
    System.out.println(d.name);
    System.out.println(d.colour);
    System.out.println(d.cost);
  }
}
```

O/p:- Bruno
Brown
30000



# VARIABLES:

```
class Demo1
{
 public static void main(String args[])
 {
   int a;
   float b;      || Local variables
   String c;
   boolean d;
   System.out.println(a);
   System.out.println(b);
   System.out.println(c);
   System.out.println(d);
 }
}
```

O/p:- error: variable might not be initialized

```
class Demo
{
  int a;
  float b;  || Instance variables
  String c;
 boolean d;
}
class launchDemo
{
  public static void main(String args[])
  {
    Demo d=new Demo();
    System.out.println(d.a);
    System.out.println(d.b);
    System.out.println(d.c);
    System.out.println(d.d);
  }
}
```

O/p:- 0
0.0
null
false

→ Local variables are those variables which are declared inside the method.

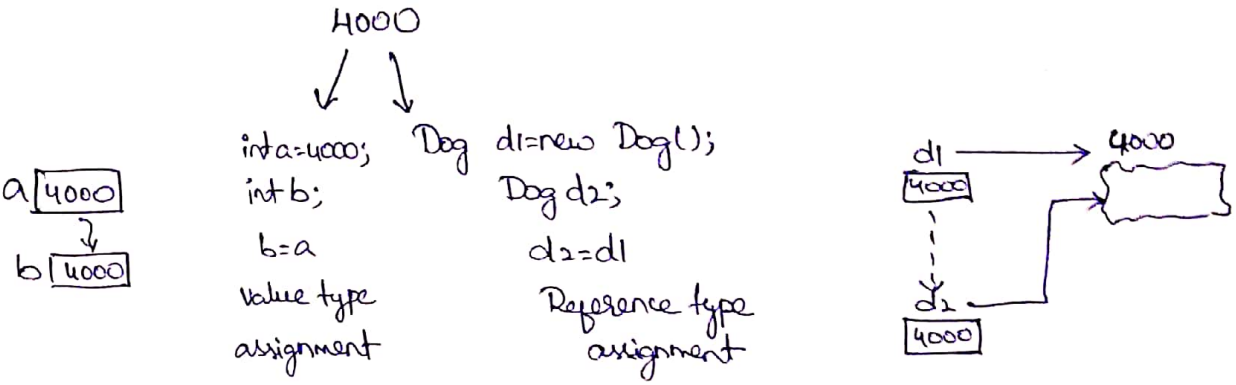
→ Instance variables are those variables which declared with in class directly out of method.

27/6/19

Difference between local variables and instance variables

| Local variables | Instance variables |
|---|---|
| (1) Local variables are those variables which are declared within a class ~~outside~~ inside a method. | (1) Instance variables are those variables which are declared directly inside a class. |
| (2) Memory for local variables will be allocated on stack segment. | (2) Memory for instance variables will be allocated on heap segment. |
| (3) Local variables cannot be used without intialization. | (3) Instance variables can be used without initialization. |
| (4) Local variables will not get intialized with default values. | (4) Instance variables get intialized with instances values. |
| (5) The memory for local variables will be deallocated once the control leaves the method. | (5) The memory for instance variables will be deallocated once the object becomes garbage object. |

## Value type argument & reference type assignment:-




## Methods:-

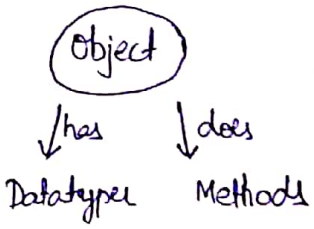
There are four types of methods

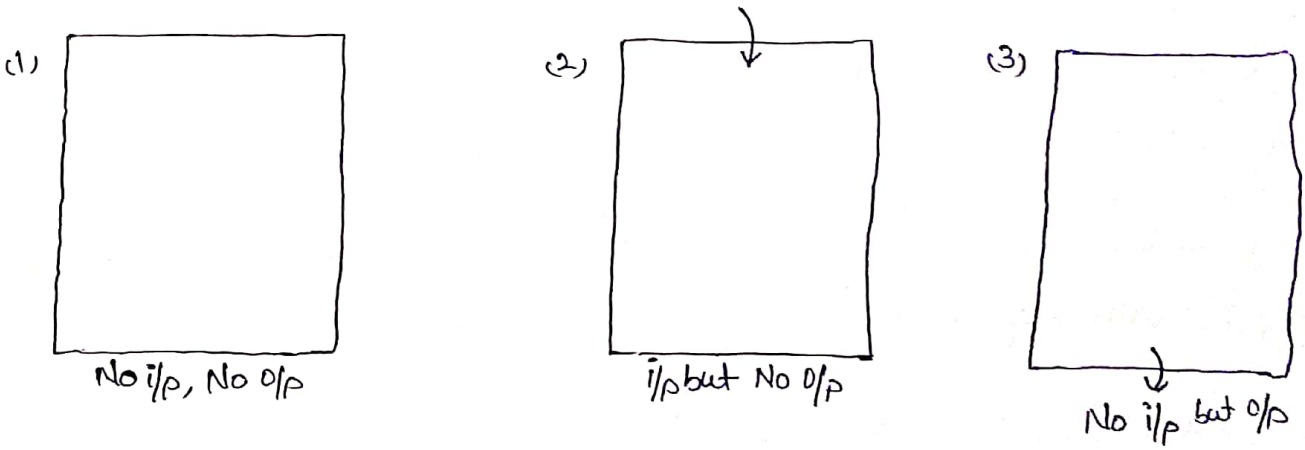
(1) No i/p , No o/p

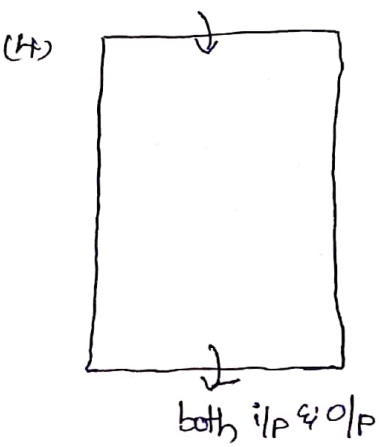
(2) i/p but no o/p

(3) No i/p but o/p

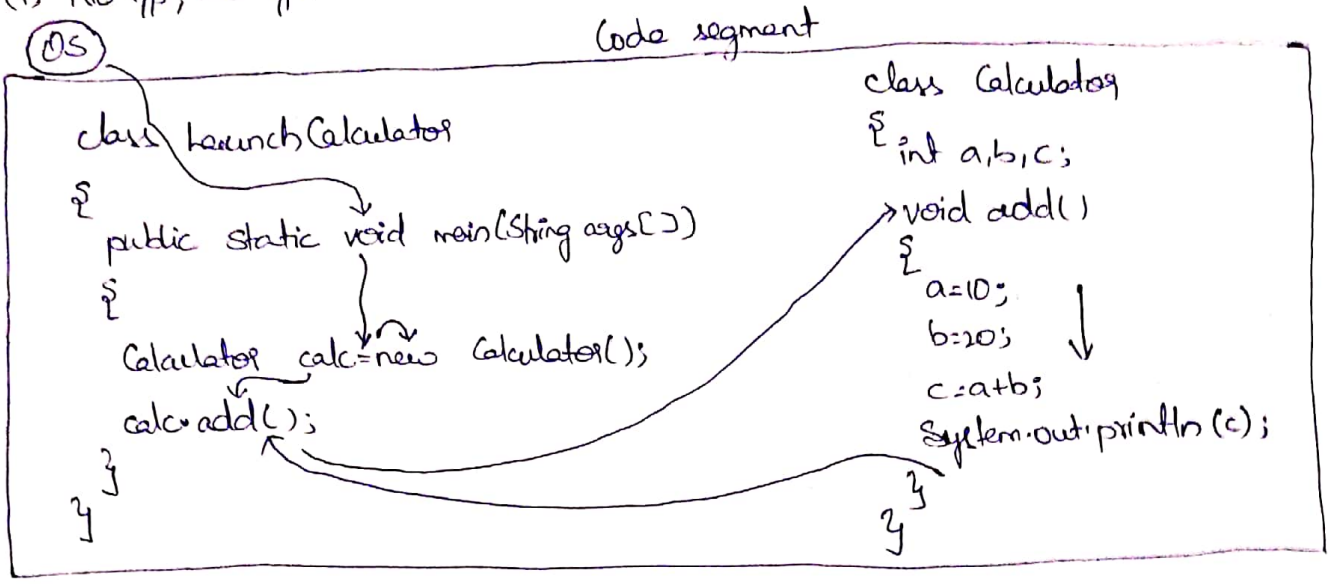
(4) both i/p & o/p



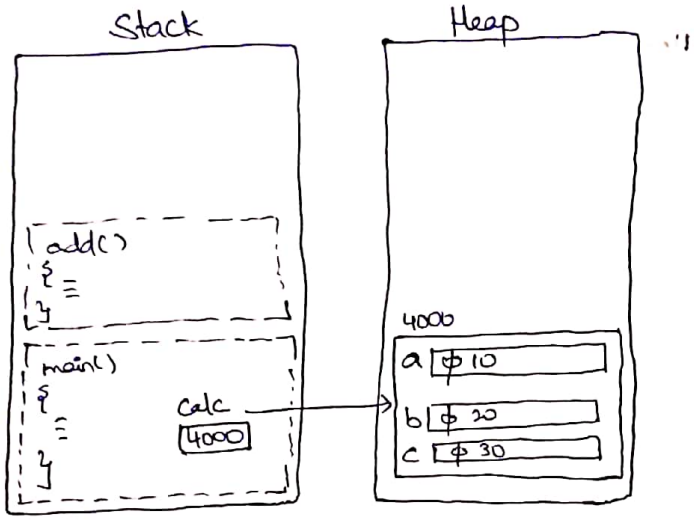



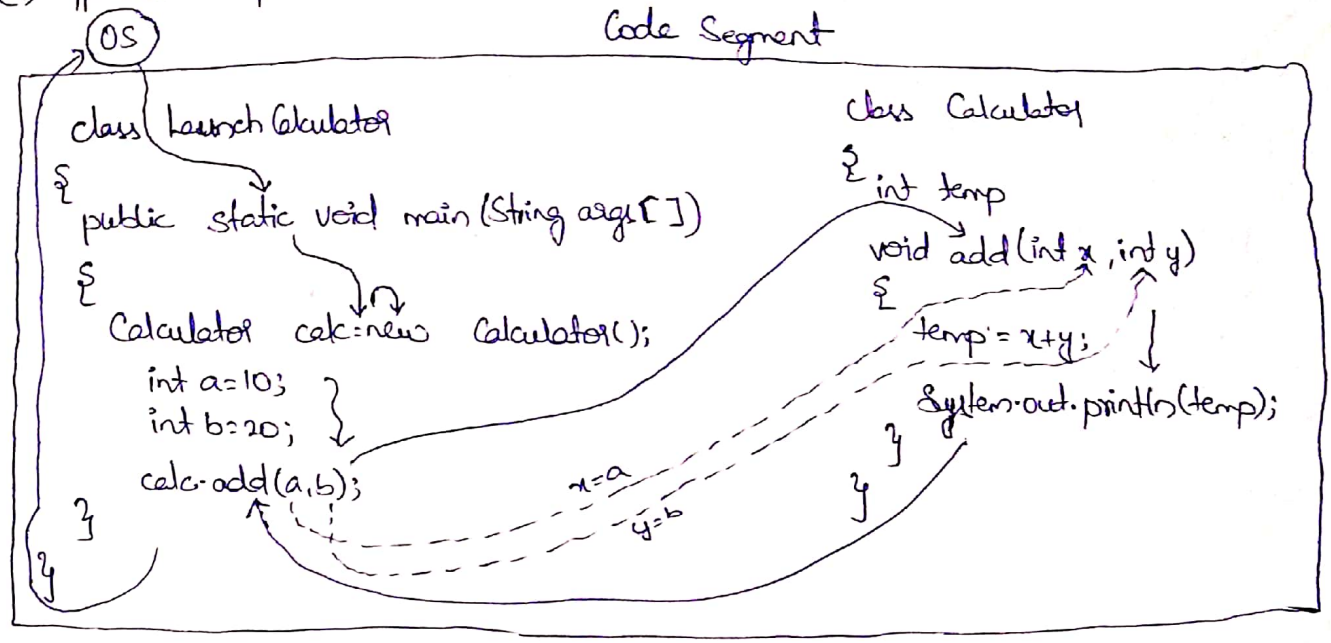
(1) No i/p, No o/p:



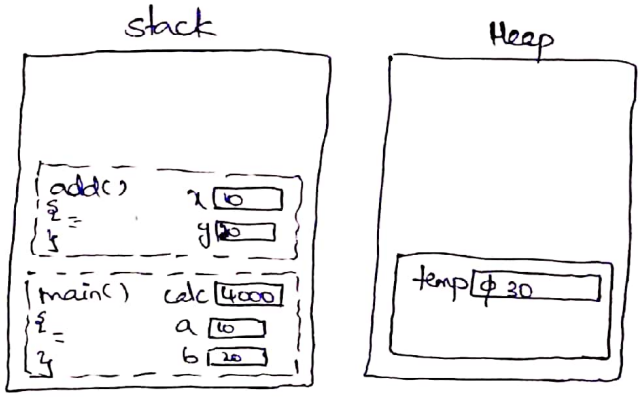

O/p: 30



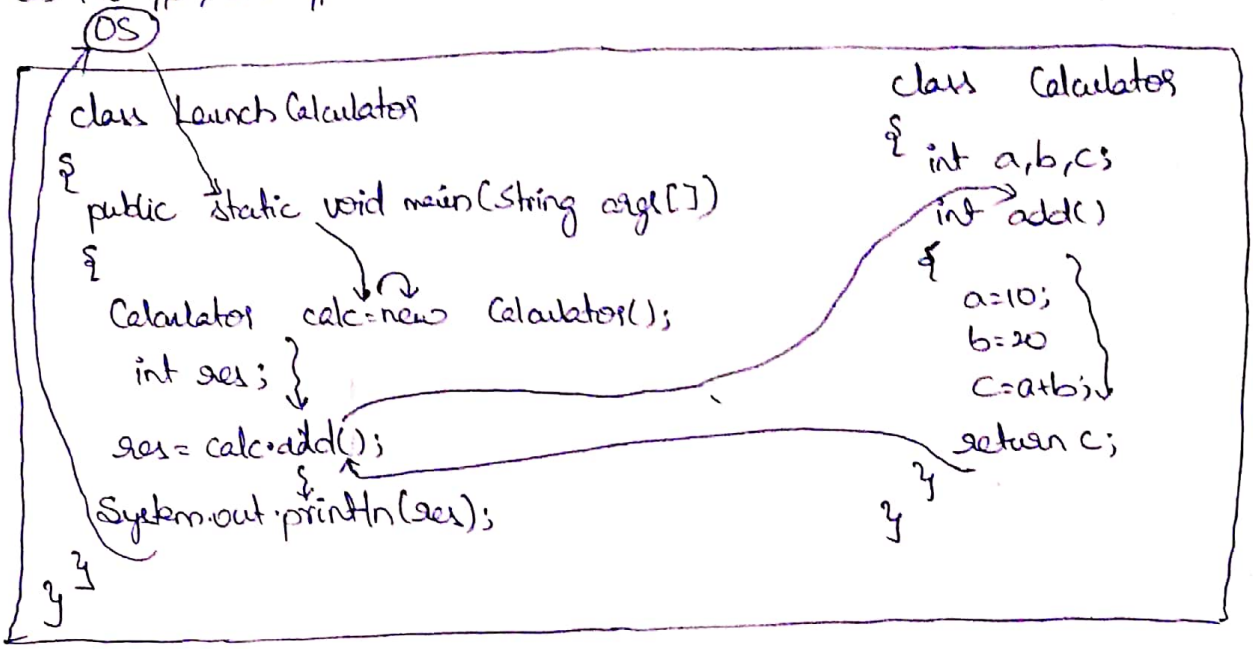
(2) i/p but no o/p :-



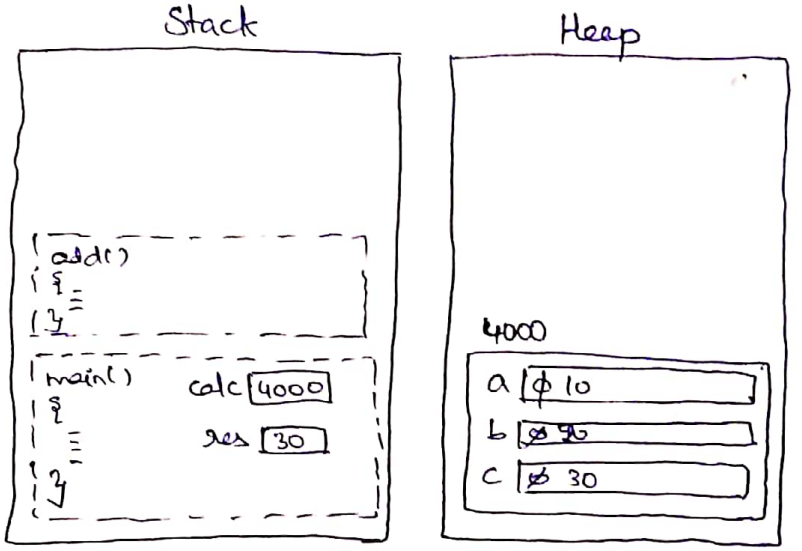

o/p :- 30



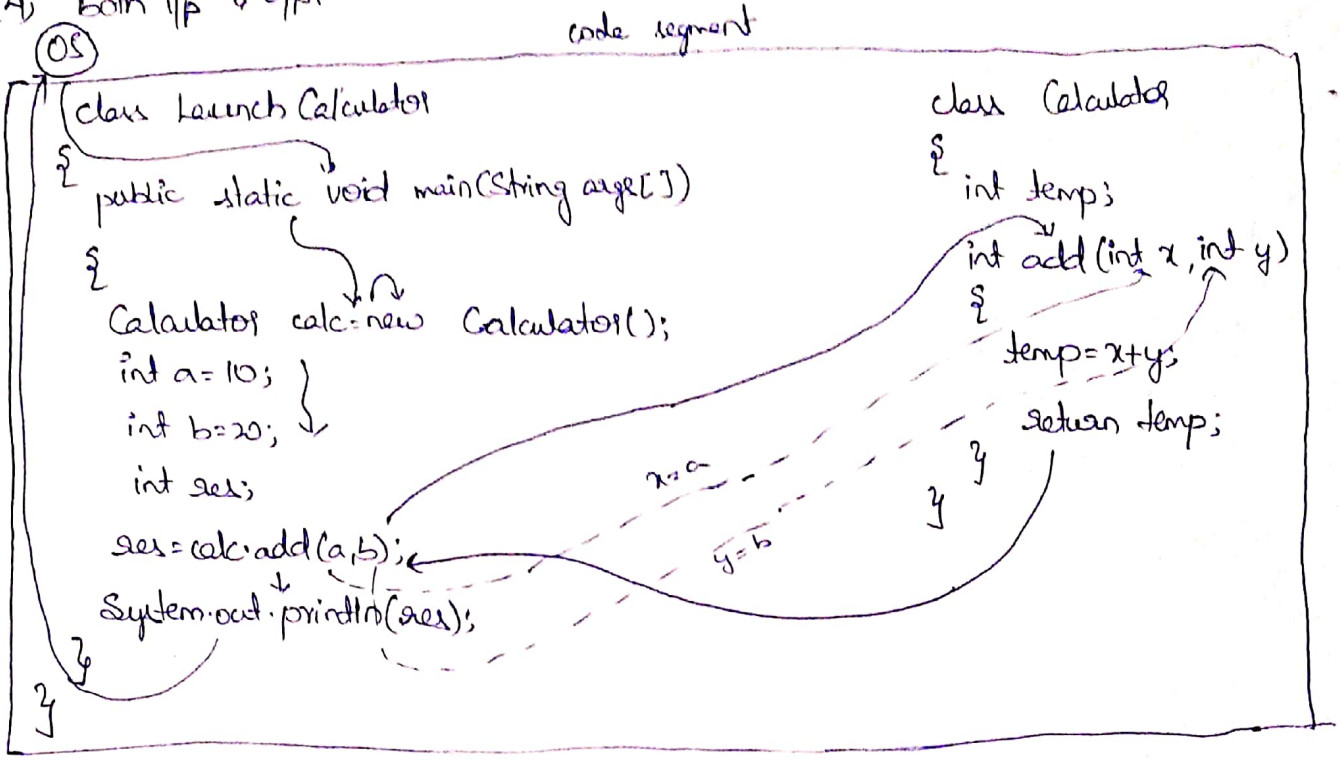
(3) No i/p, but O/p:-



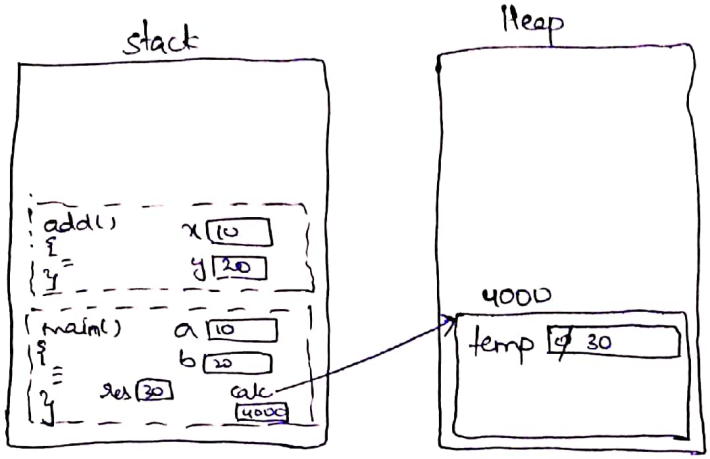

O/p:- 30



(4) both i/p & o/p:-




o/p:- 30



28/7/19

Method overloading:

Method overloading is a process of having multiple methods with same name.

| Virtual | Polymorphism |
|---|---|
| false | 1:M |

```
class Calculator
{
 int add(int x, int y)
 {
  return x+y;
 }
 float add(int x, float y)
 {
  return x+y;
 }
 float add(float x, float y)
 {
  return x+y;
 }
 double add (int x, double y)
 {
  return x+y;
 }
 int add(int x, int y, int z)
 {
  return x+y+z;
 }
 double add(int x, float y, double z)
 {
  return x+y+z;
 }
 float add (int 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, double z)
 {
  return x+y+z;
 }
```

```
double add (int x, int y, double z)
{
  return x+y+z;
}
double add (int x, double y, double z)
{
  return x+y+z;
}
float add (float x, float y)
{
  return x+y;
}
}
class LaunchCalculator
{
  public static void main(String args[])
  {
    int a=10, b=20, c=30;
    float p=10.5f, q=20.5f, r=30.5f;
    double m=15.5, n=25.5, o=35.5;
    Calculator calc=new Calculator();
    calc.add(a,b);
    calc.add(a,p);
    calc.add(a,p,m);
  }
}
```

→ In method overloading, you can have multiple methods with same name, same no. of parameters, same datatype of parameters however order of datatypes should be different.

→ Method overloading follows virtual polymorphism because programmer will be under the illusion that same/single method is performing multiple task but in reality it won't be same.

→ In method overloading return type doesn't play any role.

```
class Calculator
{
  void add(int x, float y)
  {
    float temp=x+y;
    System.out.println(temp);
  }
  float add(int x, float y)
  {
    float temp=x+y;
    return temp;
  }
}
class LaunchCalculator
{
  public static void main(String args[])
  {
    Calculator calc=new Calculator();
    calc.add(10,20.5f)
  }
}
```

## JAVA INSTALLATION (JDK SETUP):-

Don't touch system variables.

In user variables

New↵

variable name:- JAVA_HOME

c:programfiles:java:jdk1.8

copy path

variable value: [ ] ↳Paste path

New↵

variable name: Path

variable value: [ ]

↳Paste path, put \bin

1/7/19

Method overloading with numeric proportion:-

```
class Calculator
{
  float add(int x,int y)
  {
    return x+y;
  }
  float add(int x,float y,int z)
  {
    return x+y+z;
  }
}

class LaunchCalculator
{
  public static void main(String args[])
  {
    Calculator calc=new Calculator();
    System.out.println(calc.add(10,20));
  }
}
```

O/p:- 30.0

→ If multiple methods are promoting numeric type promotion it leads to ambiguity.

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

```
class LaunchCalculator
{
  public static void main(String args[])
  {
    Calculator calc=new Calculator();
    System.out.println(calc.add(10,20));
  }
}
```

O/p:- Error

reference add is ambigious

Converting strings to tokens using inbuilt class:-

```
import java.util.StringTokenizer;
class Demo
 {
   public static void main(String args[])
   {
    StringTokenizer s=new StringTokenizer("ABC FOR TECHNOLOGY TRAINING",
                                          " ");
    while(s.hasMoreTokens()==true)
    {
     System.out.println(s.nextToken());
    }
   }
 }
```

O/p:- ABC
FOR
TECHNOLOGY
TRAINING

## Encapsulation:

→ Private is the keyword used for providing security and preventing direct access.

→ Getter, setter methods are user defined methods.

```
class Books
{
  private int pg_no;
  void setData(int x)
  {
    if(x>0)
    {
      pg_no=x;
    }
    else
    {
      System.out.print("Invalid use");
      System.exit(0);
    }
  }
  int getData()
  {
    return pg_no;
  }
}
class LaunchBook
{
  public static void main(String args[])
  {
    Books b=new Books();
    b.setData(100);
    System.out.println(b.getData());
  }
}
```

→ Encapsulation is a process of providing security to most important component of object.

→ Encapsulation is preventing direct access and providing controlled access to object.

→ Encapsulation can be achieved by using private members, setter's and getter's.

2/7/19

```
class Dog
{
  private String name;
  private String colour;
  private int cost;

  void setData(String x, String y, int z)
  {
    name = x;
    colour = y;
    cost = z;
  }
  String getName()
  {
    return name;
  }
  String getColour()
  {
    return colour;
  }
  int getCost()
  {
    return cost;
  }
}
class launchDog
{
  public static void main(String args[])
  {
    Dog d = new Dog();
    d.setData("Jimmy", "Black", 1000);
    System.out.println(d.getName());
    System.out.println(d.getColour());
    System.out.println(d.getCost());
  }
}
```

O/P:- Jimmy
Black
1000

→ Constructor is a special type of setter which is having same name as of class name and they won't be any explicit return type

```
class Dog
{
  private String name;
  private String colour;
  private int cost;

  Dog(String x, String y, String z)
  {
    name = x;
    colour = y;
    cost = z;
  }
  String getName()
  {
    return name;
  }
  String getColour()
  {
    return colour;
  }
  int getCost()
  {
    return cost;
  }
class LaunchDog
{
  public static void main(String args[])
  {
    Demo d = new Demo("Jimmy", "Black", 1000);
    System.out.println(d.getName());
    System.out.println(d.getColour());
    System.out.println(d.getCost());
  }
}
```

O/P: Jimmy
Black
1000

```
class Dog
{
  private String name;
  private String colour;
  private   int  cost;

  Dog(String name, String colour, int cost)
  {
    name=name;
    colour= colour;
     cost = cost;
  }
  String getName()
  {
    return name;
  }
  String getColour()
  {
    return colour;
  }
  int getCost()
  {
    return cost;
  }
}
class LaunchDog
{ public static void main(String args[])
  {
    Dog d=new Dog("Jimmy","Black", 1000);
    System.out.println(d.getName());
    System.out.println(d.getColour());
    System.out.println(d.getCost());
  }
}
```

o/p: null
null
0

## Shadowing problem:-

Whenever local variables and instance variables names matches shadowing problem occurs.

Note:-

(1) Shadowing problem can be overcome by this keyword.

(2) This keyword refers to instance variable.

```
class Dog
{
  private String name;
  private String colour;
  private int cost;

  Dog(String name, String colour, int cost)
  {
    this.name=name;
    this.colour=colour;
    this.cost=cost;
  }
  String getName()
  {
    return name;
  }
  String getColour()
  {
    return colour;
  }
  int getCost()
  {
    return cost;
  }
}
```

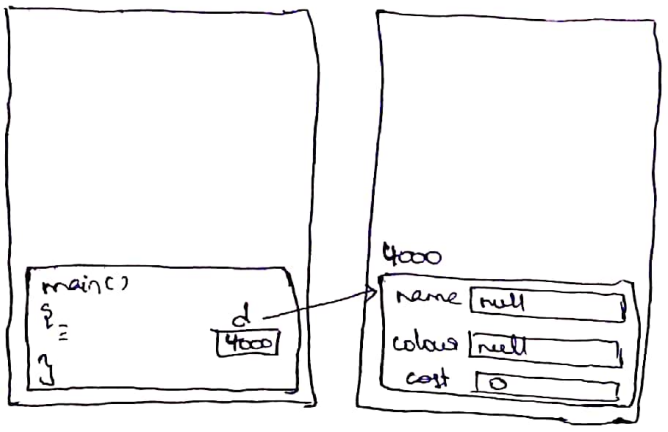
```
class launchDog
{
  public static void main(String args[])
  {
    Dog d=new Dog("Jimmy","Black",1000);
    System.out.println(d.getName());
    System.out.println(d.getColour());
    System.out.println(d.getCost());
  }
}
```

O/P: Jimmy
Black
1000

```
Class Dog extends Object
{
  private String name;
  private String colour;
  private int cost;
  Dog()
  {
    super();
  }
  String getName()
  {
    return name;
  }
  String getColour()
  {
    return colour;
  }
  Int getCost()
  {
    return cost;
  }
}
```

```
class LaunchDog
{
  public static void main(String args[])
  {
    Dog d=new Dog();
    System.out.println(d.getName());
    System.out.println(d.getColour());
    System.out.println(d.getCost());
  }
}
```

O/p: null
null
0




3/7/19

→ Automatically a java compiler will include a default constructor within a class. Inside a constructor there would be a super method call and duty of super method call is to take the control from child class to parent class constructor.

→ Object class is the parent class of all the classes in java.

```
class Dog extends Object
{
  private String name;
  private String colour;
  private int cost;
    Dog()
    {
      super();
    }
    String getName()
    {
      return name;
    }
    String getColour()
    {
      return colour;
    }
     int getCost()
     {
       return cost;
     }
}
class LaunchDog
 {
   public static void main(String args[])
   {
     Dog d=new Dog("Jimmy","Black",10000);
     System.out.println(d.getName());
     System.out.println(d.getColour());
     System.out.println(d.getCost());
   }
 }
```

O/p: error: constructor Dog in class Dog cannot be applied to given types;

required: no arguments

NOTE:-

→ A java compiler would always include default (zero-parameterized) constructor irrespective of type of constructor called by the programmer.

```
class Dog extends Object
{
  private String name;
  private String colour;
  private int cost;

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

  String getName()
  {
    return name;
  }
  String getColour()
  {
    return colour;
  }
  String getCost()
  {
    return cost;
  }
}
```

```
(OS)
class LaunchDog
{
  public static void main(String args[])
  {
    Dog d1=new Dog("Jimmy","Black",10000);
    System.out.println(d1.getName());
    System.out.println(d1.getColour());
    System.out.println(d1.getCost());

    Dog d2=new Dog()
    System.out.println(d2.getName());
    System.out.println(d2.getColour());
    System.out.println(d2.getCost());
  }
}
```

O/p: Error

```
class Dog extends Object
{
    private String name;
    private String colour;
    private int cost;

    Dog(String name, String colour, int cost)
    {
      super();
      this.name=name;
      this.colour=colour;
       this.cost=cost;
    }
    String getName()
    {
      return name;
    }
    String getColour()
    {
      return colour;
    }
    int getCost()
    {
      return cost;
    }
}
```

```
class LaunchDog
{
  public static void main(String args[])
  {
   Dog d1=new Dog("Jimmy","Black",10000);
   System.out.println(d1.getName());
  System.out.println(d1.getColour());
  System.out.println(d1.getCost());

   Dog d2=new Dog();
   System.out.println(d2.getName());
   System.out.println(d2.getColour());
   System.out.println(d2.getCost());
  }
}
```

O/p: Jimmy
Black
10000
null
null
0

Constructor chaining:

→ It is a process of calling child class constructor calling parent class constructor.

→ It can be achieved by super method call.

Local chaining:

→ It is a process of one constructor calling another constructor present within same class. Local chaining can be achieved by using this method call.

04/7/19

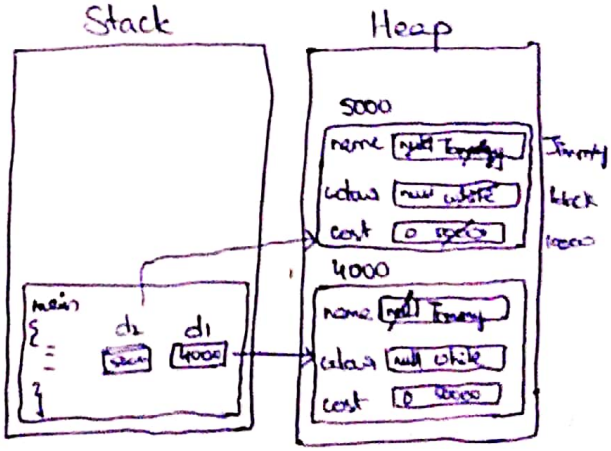
OS

```
class Dog extends Object
{
 private String name;
 private String colour;
 private int cost;

 Dog()
 {
  super();
  this.name="Tommy";
  this.colour="White";
  this.cost=5000;
 }

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

 String getName()
 {
  return name;
 }
 String getColour()
 {
  return colour;
 }
 int getCost()
 {
  return cost;
 }
}
```

```
class Object
{
 Object()
 {
 }
}
```

local chaining

```
class LaunchDog
{
 public static void main(String args[])
 {
  Dog d1=new Dog();
  System.out.println(d1.getName());
  System.out.println(d1.getColour());
  System.out.println(d1.getCost());
  Dog d2=new Dog("Jimmy","Black",10000);
  System.out.println(d2.getName());
  System.out.println(d2.getName());
  System.out.println(d2.getCost());
 }
}
```




O/p:- Tommy
white
5000
Jimmy
Black
10000

05

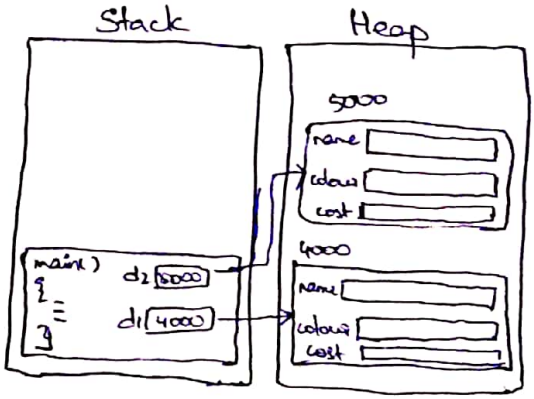
```
class Launch Dog
{
 public static void main(String args[])
 {
  Dog d1=new Dog();
  System.out.println(d1.getName());
  System.out.println(d1.getColour());
  System.out.println(d1.getCost());

  Dog d2=new Dog("Jimmy","Black",10000);
  System.out.println(d2.getName());
  System.out.println(d2.getColour());
  System.out.println(d2.getCost());
 }
}
```

O/P:- Tommy
white
50000

Jimmy
Black
10000

```
class Dog extends Object
{
 private String name;
 private String colour;
 private int cost;

 Dog()
 {
  this("xoxo");
  this.name="Tommy";
  this.colour="white";
  this.cost=50000;
 }

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

 String getName()
 {
  return name;
 }
 String getColour()
 {
  return colour;
 }
 Int getCost()
 {
  return cost;
 }
 Dog(String name)
 {
  super();
  this.name=name;
 }
}
```

```
class Object
{
 Object()
 {
 }
}
```



5/7/19

NOTE:

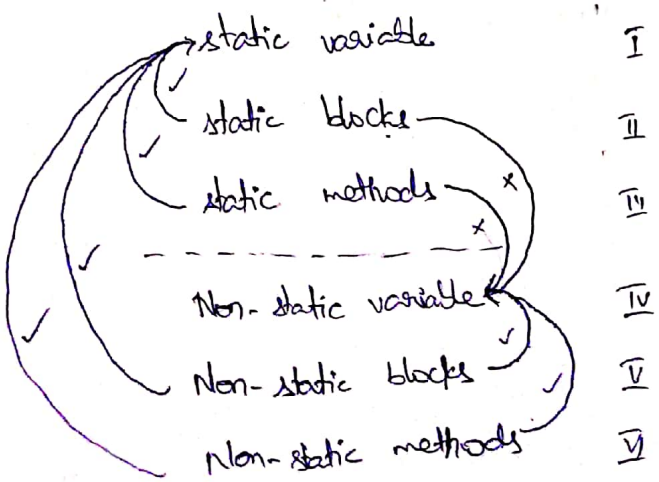
1) Even though constructor donot have any explicit return type, it will return address of an object.

## Static keyword:-

There are different types of elements in static

1) Static variable
2) Static block
3) Static methods

```
class Demo
{
  static variable; (1)
  Non-static variable; (4)
  static
  {
    = (2)
  }
  {
    = (5)
  }

  static method
  {
    = (3)
  }
  Non-static methods
  {
    = (6)
  }
}
```



→ The order execution of program will start from

(1) static variable

(2) static block

(3) static method

(4) Non-static variable

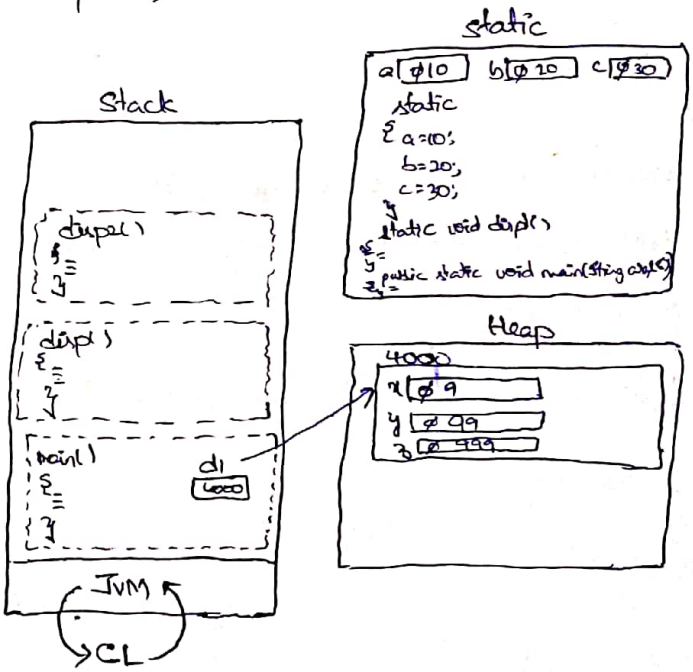
(5) Non-static block

(6) Non-static methods

```
class Demo
{
  static int a,b,c;
   int x,y,z;
  static
  {
    a=10;
    b=20;
    c=30;
  }
  {
    x=9;
    y=99;
    z=999;
  }
  static void disp1()
  {
   System.out.println(a);
   System.out.println(b);
   System.out.println(c);
  }
   void disp2()
  {
    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
    System.out.println(x);
    System.out.println(y);
    System.out.println(z);
  }
}
```

```
class LaunchDemo
{
  public static void main(String args[])
  {
    Demo.disp1();
    Demo d1=new Demo()
        d1.disp2();
  }
}
```

```
O/P: 10
     20
     30
     10
     20
     30
      9
     99
    999
```



Class Loader(CL):-

It scans the whole code in the code segment completely to check is there any static elements in code, if there are any static elements are there it will loaded into static segment.

NOTE: 1) Code segment and static segments are fixed choice segments

8/7/19 2) Stack segment and heap segments are variable size segments.

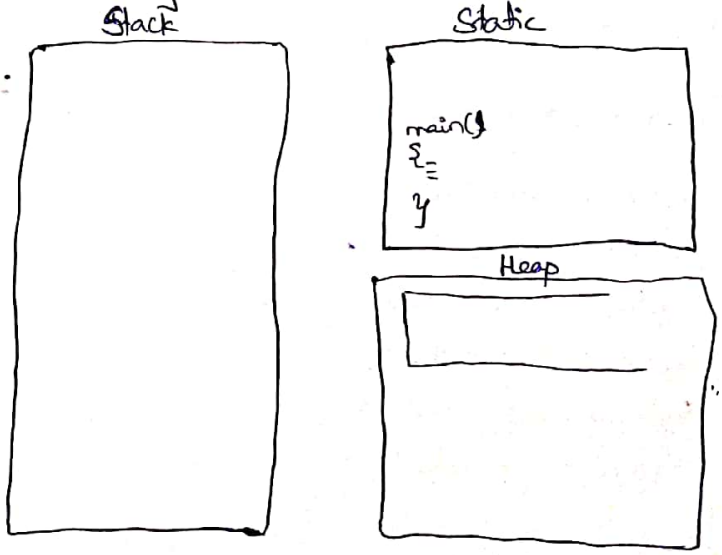
Application of static variable:-

(1) Without using static variable:

```
import java.util.Scanner;
class Fsimp
{
 float p;
 float t;
 float r;
 float si;
 void input()
 {
  Scanner scan=new Scanner(System.in);
  System.out.println("Enter principle amount");
  p=scan.nextFloat();
  System.out.println("Enter time duration");
  t=scan.nextFloat();
  r=2.5f;
 }
 void compute()
 {
  si=(p*t*r)/100;
 }
 void disp()
 {
  System.out.println(si);
 }
}
```

```
class LaunchFsimp
{
 public static void main(String args[])
 {
  Fsimp f1=new Fsimp();
  Fsimp f2=new Fsimp();
  Fsimp f3=new Fsimp();
  f1.input();
  f2.input();
  f3.input();
  f1.compute();
  f2.compute();
  f3.compute();
  f1.disp();
  f2.disp();
  f3.disp();
 }
}
```

O/p: 25.0
100.0
225.0

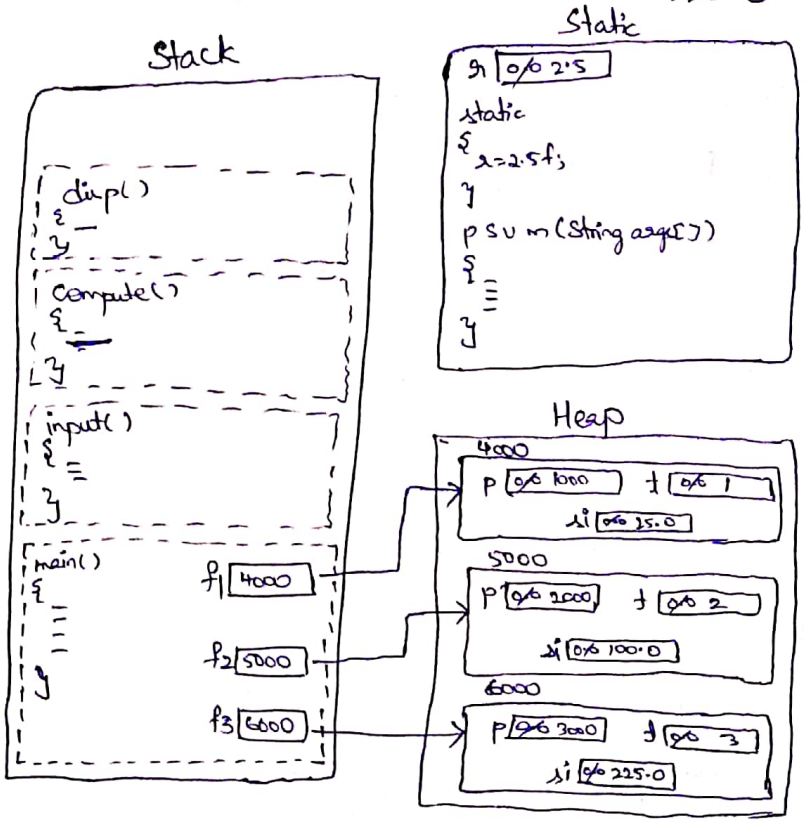


(2) with using static variable:-

```
import java.util.Scanner;
class Farmer
{
 float p;
 float t;
 static float r;
 float si;

 static
 {
  r=2.5f;
 }
 void input()
 {
  Scanner scan=new Scanner(System.in);
  System.out.println("Enter principle amount:");
  p=scan.nextfloat;
  System.out.println("Enter time duration:");
  t=scan.nextFloat;
 }
 void compute()
 {
  si=(p*t*r)/100;
 }
 void disp()
 {
  System.out.println(si);
 }
}
```

```
class LaunchFarmer
{
 public static void main(String args[])
 {
  Farmer f1=new Farmer();
  Farmer f2=new Farmer();
  Farmer f3=new Farmer();
  f1.input();
  f2.input();
  f3.input();
  f1.compute();
  f2.compute();
  f3.compute();
  f1.disp();
  f2.disp();
  f3.disp();
 }
}
```

O/P:- 25.0
100.0
225.0



9/7/19 NOTE:

(1) In version 1 program the data of interest is common for all the instance object's ∴ a variable which is common for all the object should be declared as static.

(2) In version 1 program memory is not efficiently used.

NOTE:

(1) Whenever there is a common that has to be shared among all/multiple objects then we should declare that kind of variable as static variable.

Difference static variable and Instance variable:-

| STATIC VARIABLE | INSTANCE VARIABLE |
|---|---|
| (1) Memory for static variable will be allocated on static segment. | (1) Memory for instance variables will be allocated on heap segment. |
| (2) Static variables will be initialized by JVM to default values. | (2) It will be initialized by constructor to default values. |
| (3) Common copies for all objects. | (3) One copy for one object. |
| (4) Static variables can be accessed by using class name. | (4) Instance variables cannot be accessed by using class name. |
| (5) Static variables are created using static keyword. | (5) Instance variables cannot be used with static keyword. |
| (6) Memory for static variables will not be allocated using garbage collector. | (6) Memory will be deallocated using garbage collector. |

## Application of static block:

```
class Demo
{
  static
  {
    System.out.println("ABC");
  }
  public static void main(String args[])
  {
    System.out.println("spiders");
  }
}
```

O/p:- ABC
spiders

NOTE:

(1) Wherever there are set of statements that has to executed before before execution of main method, we will use static block.

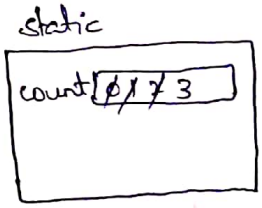
10/7/19

## Application of non-static block:

Write a java program to count the number of objects created. using static variable and without using non-static block.

```
class Dog
{
  String name;
  String colour;
  int cost;
  static int count;

  Dog()
  {
    ++count;
  }
  Dog(String name, String colour, int cost)
  {
    this.name = name;
    this.colour = colour;
```

```
        this.cost= cost;
        ++ count;
    }
}
class LaunchDog
{
    public static void main(String args[])
    {
        System.out.println(Dog.count);
        Dog d1=new Dog();
        Dog d2=new Dog();
        System.out.println(Dog.count);
        Dog d3= new Dog();
        System.out.println ("Jimmy","Black",10000);
        System.out.println(Dog.count);
    }
}
```




O/p: 0
2
3

The above program can also be written by non-static block. A non-static block will get executed immediately after super method call.

Using static variables and non-static block:-

```
class Dog
{
  String name;
  String colour;
   int cost;
   static int count;

   {
    ++count;
   }

   Dog()
   { super();
   }

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

class LaunchDog
{
 public static void main(String args[])
 {
   System.out.println(Dog.count);
   Dog d1=new Dog();
   Dog d2=new Dog();
   System.out.println(Dog.count);
   Dog d3=new Dog("Jimmy", "Black", 10000);
   System.out.println(Dog.count);
 }
}
```


O/P:- 3

11/7/19

```
class Demo extends Object
{
   {
     System.out.println("Within non-static block");
   }
   Demo()
   {
     System.out.println("Inside a constructor");
   }
}

class LaunchDemo
{
  public static void main(String args[])
  {
    Demo d=new Demo();
  }
}
```

O/p:- Within non-static block
Inside a constructor

| Static Method | Non-static Method |
|---|---|
| (1) Static methods are called as utility methods (General methods). | (1) Non-static methods are called as specific methods. |
| (2) Static methods can be invoked without creating an object. | (2) They cannot be invoked without creating object. |
| (3) Ex:- currentThread()<br>binarySearch() | (3) Ex:- NumForGuesses()<br>numForPlayers() |

→ Whenever common copy has to shared among multiple objects then we should declared as static

→ If every object requires a separate copy (individual copy) then we should declare these variables non-static.

→ If there are few statements which has to be executed before main method execution then we should use static block.

→ If there are few statements that compulsory has to get executed for all the objects then we should make use of non-static block.

→ If there are few methods which should be invoked after object creation then we make use of non-static methods.

→ If there are few methods which can be invoked without creation of object then we should declare method as static.

---

→ If there are multiple blocks then execution of non-static blocks will be done as per the order of occurance.

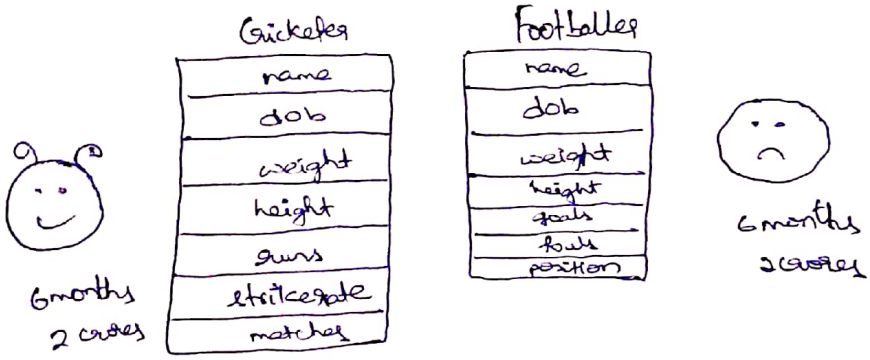
```
class Demo
{
  {
    System.out.println("Within non-static block1");
  }
  {
    System.out.println("Within non-static block2");
  }
  Demo()
  {
    System.out.println("inside a constructor");
  }
}
```

```
class LaunchDemo
{
  public static void main(String args[])
  {
    Demo d=new Demo();
  }
}
```

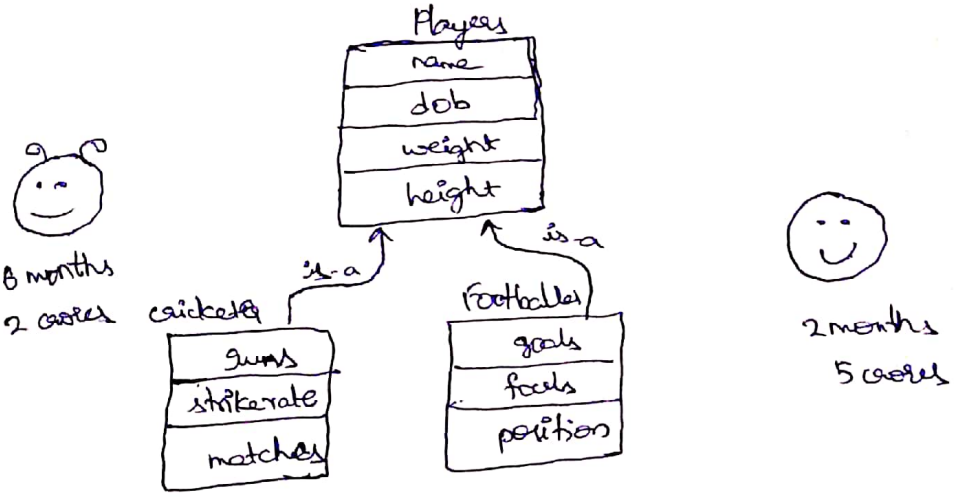
O/p:- Within non-static block1
Within non-static block2

## Inheritance :-

Company I (no inheritance)




Company II (inheritance)




12/7/19

Inheritance refers to writing a program of hierarchy of classes. Inheritance promotes code reusability, reduce the coding time and also it brings profit to the company.

Inheritance promotes is-a relationship, in order to achieve inheritance we have to make use of extends keyword.

## Rules of inheritance:

(1) Private members are not inherited in java, this is to promote encapsulation

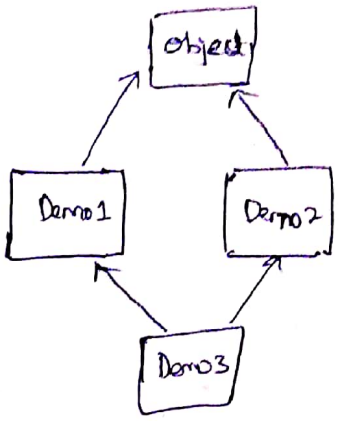
```
class You
{
  private int accno;
  private int pwd;

  You()
  {
    accno=1111;
    pwd=2222;
  }
}

  class Friend extends You
  {
    Friend()
    {
      accno=3333;
      pwd=4444;
    }
    void disp()
    {
      System.out.println(accno);
      System.out.println(pwd);
    }
  }

    class LaunchFriend
    {
      public static void main(String args[])
      {
        Friend f=new Friend();
        f.disp()
      }
    }
```

O/p: 3333 Error
4444

(2) Multiple inheritance is not permitted in java.

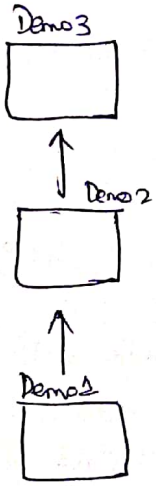
Multiple inheritance leads to diamond shape and diamond shape problem will results in ambiguity.

O/p: error: '{' expected



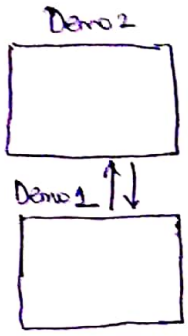

```
class Demo1
{

}

class Demo2
{

}

class Demo3 extends Demo1,Demo2
 {
   public static void main(String args[])
   {

   }
 }
```

(3) Eventhough multiple inheritance is not permitted in java multilevel inheritance is permitted




```
class Demo1 extends Demo2
{
 }
 class Demo2 extends Demo3
 {
 }
 class Demo3
 {
   public static void main(String args[])
   {
   }
 }
```

4) Cyclic inheritance is not permitted in java.



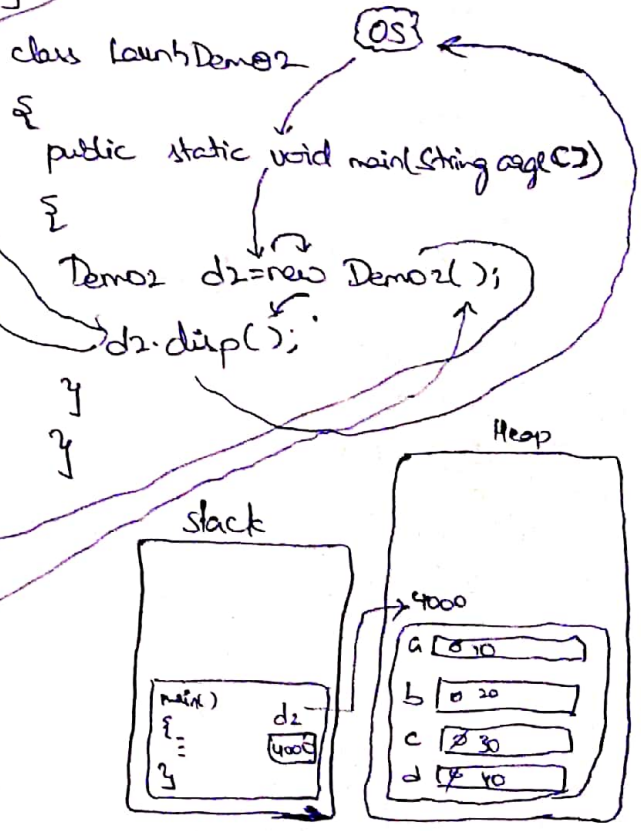

15/7/19

Execution of constructors in case of inheritance:-

Program 1:-

```
class Demo1 extends Object
{
 int a;
 int b;
 Demo1()
 {
  super();
  a=10;
  b=20;
 }
 Demo1(int m, int n)
 {
  super();
  a=m;
  b=n;
 }
}
class Demo2 extends Demo1
{
 int c;
 int d;
 Demo2()
 {
  super();
  c=30;
  d=40;
 }
 Demo2(int p, int q)
 {
  super();
  c=p;
  d=q;
 }
 void disp()
 {
  System.out.print(a);
  System.out.print(b);
  System.out.print(c);
  System.out.print(d);
 }
}
class LaunchDemo2
{
 public static void main(String args[])
 {
  Demo2 d2=new Demo2();
  d2.disp();
 }
}
```

class Object
{
 Object()
 {
 }
}



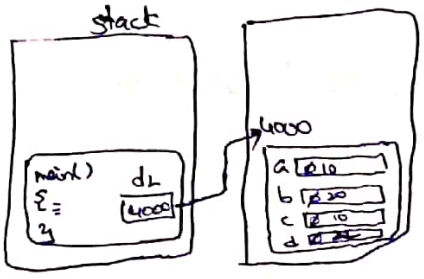

O/P:- 10
20
30
40

Whenever object of child class is created the memory is not only allocated for child class but it is also allocated for parent class therefore creating an object of child class is equivalent creating an object for parent class.

Program 2:-

```
class Demo1 extends Object
{
  int a;
  int b;

  Demo1()
  {
   super();
   a=10;
   b=20;
  }

  Demo1(int m, int n)
  {
   super();
   a=m;
   b=n;
  }
}
class Demo2 extends Demo1
{
  int c;
  int d;
  Demo2()
  {
   super();
    c=30;
   d=40;
  }
  Demo2(int p, int q)
  {
   super();
    c=p;
    d=q;
  }
  void disp()
  {
   System.out.println(a);
   System.out.println(b);
   System.out.println(c);
   System.out.println(d);
  }
}

class LaunchDemo2
{
 public static void main(String args[])
 {
   Demo d2=new Demo2(10,22);
   d2.disp();
 }
}
```

O/P: 10
20
10
22



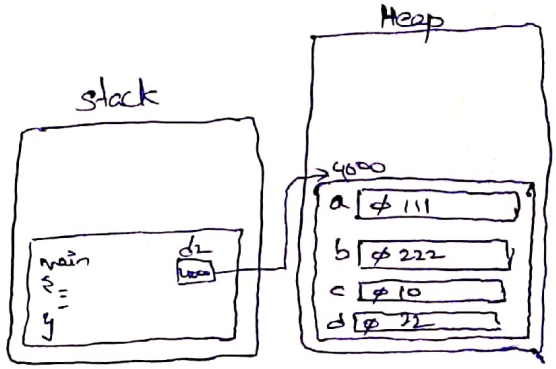
11/7/19

Whenever a parameterized constructor of child class is invoked but still it is the zero parameterized constructor of parent class which gets invoked.

```
class Demo1 extends Object
{
 int a;
 int b;
 Demo1()
 {
  super();
  a=10;
  b=20;
 }
 Demo1(int m, int n)
 {
  super();
  a=m;
  b=n;
 }
}
class Demo2 extends Demo1
{
 int c;
 int d;
 Demo2()
 {
  super();
  c=30;
  d=40;
 }
 Demo2(int p, int q)
 {
  super(111,222)
  c=p;
  d=q;
 }
 void disp()
 {
  System.out.println(a);
  System.out.println(b);
  System.out.println(c);
  System.out.println(d);
 }
}
```

```
class LaunchDemo
{
 public static void main(String args[])
 {
  Demo2 d2=new Demo2(10,22);
  d2.disp();
 }
}
```

O/P:- 111
222
10
22

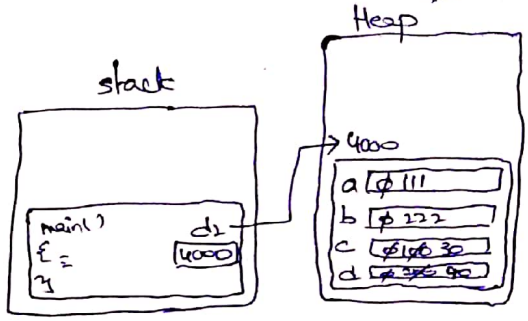


If parameterized constructor of parent class has to be invoked or executed then a super method has to included by the programmer. A programmer can include super method call with or without parameters.

```
class Demo1 extends Object
{
 int a;
 int b;

 Demo1()
 {super();
  a=10;
  b=20;
 Demo1(int m, int n)
 {super();
  a=m;
  b=n;
 }
}

class Demo2 extends Demo1
{int c;
 int d;
 Demo2()
 {
  this(100,200);
   c=30;
   d=40;
 }
 Demo2(int p,int q)
 {super(111,222);
  c=p;
  d=q;
 }
 void disp()
 {
  System.out.println(a);
  System.out.println(b);
  System.out.println(c);
  System.out.println(d);
 }
}
```

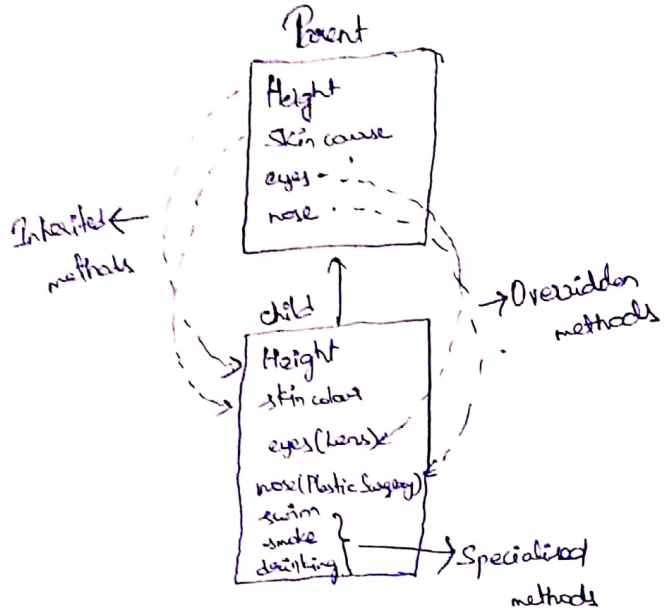
```
(OS)
class LaunchDemo
{
 public static void main(String args[])
 {
  Demo2 d2=new Demo2();
  d2.disp();
 }
}
```

O/P:- 111
222
30
40



In a program, programmer can include both super() call and this() call.

Different types of methods in inheritance:




17/7/19

Inherited methods:

Inherited methods are such methods which are inherited from parent class and are used as it is in child class without any modifications.

Overriden methods:

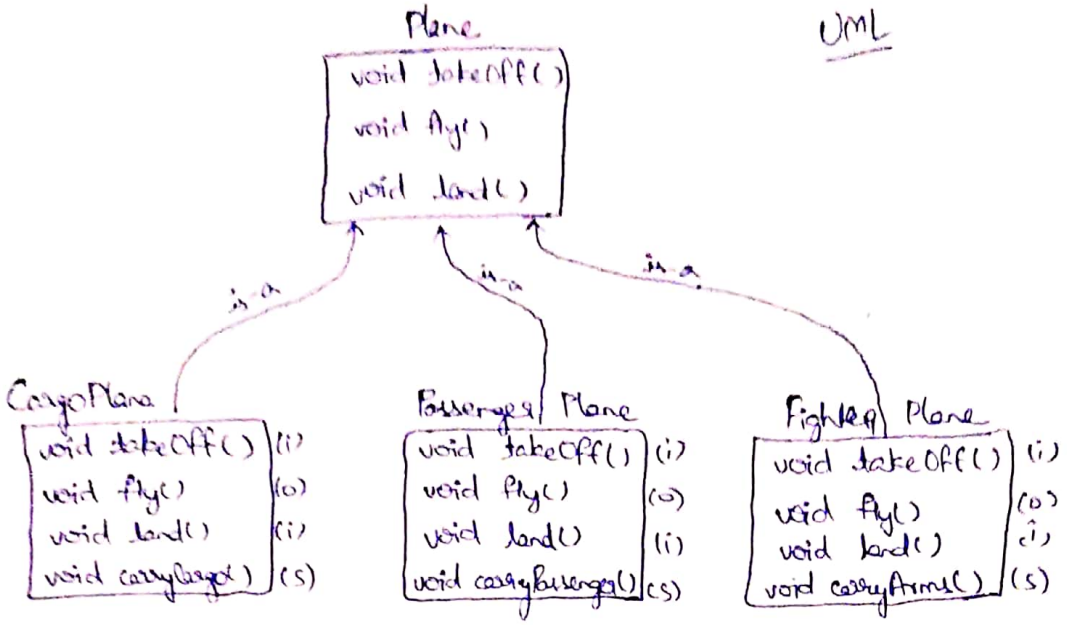
Overriden methods are such methods which are inherited from parent class and are modified to suit its requirements of child class.

Specialized methods:

Specialized methods are such methods which are not there in parent class but are present in child class.

(1)

UML




```
class Plane
{
  void takeOff()
  {
    System.out.println("Plane is taking off");
  }
  void fly()
  {
    System.out.println("Plane is flying");
  }
  void land()
  {
    System.out.println("Plane is landing");
  }
}
class CargoPlane extends Plane
{
    void fly()
    {
      System.out.println("Plane is flying at lower heights");
    }
```

```
    void CarryCargo()
    {
      System.out.println("Plane is carrying cargo");
    }
}

class PassengerPlane extends Plane
{
  void fly()
  {
    System.out.println("Plane is flying at medium heights");
  }
  void carryPassengers()
  {
    System.out.println("Plane is carrying passengers");
  }
}

class FighterPlane extends Plane
{
  void fly()
  {
    System.out.println("Plane is flying at higher heights");
  }
  void carryArms()
  {
    System.out.println("Plane is carrying arms");
  }
}
```

```
class LaunchPlane
{
 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();

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

  pp.takeOff();
  pp.fly();
  pp.land();
  pp.carryPassengers();

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

  fp.takeOff();
  fp.fly();
  fp.land();
  fp.carryArms();

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

O/p:- Plane is taking off

Plane is flying at lower heights

Plane is landing

Plane is carrying cargo.

**********

Plane is taking off

Plane is flying at medium heights

Plane is landing

Plane is carrying passengers

**********

Plane is taking off

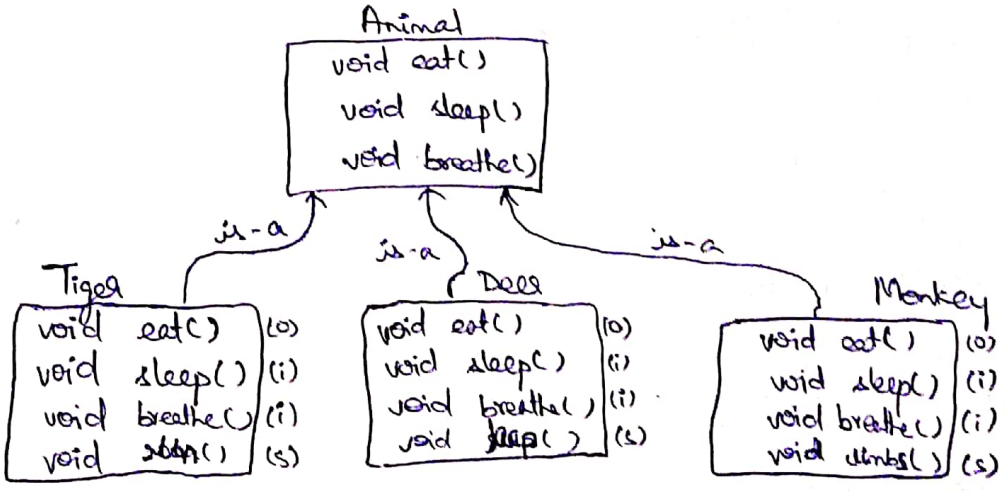
Plane is flying at highest heights

Plane is landing

Plane is carrying arms

**********

(2)




```
class Animal
{
  void eat()
  {
    System.out.println("Animals eats food");
  }
```

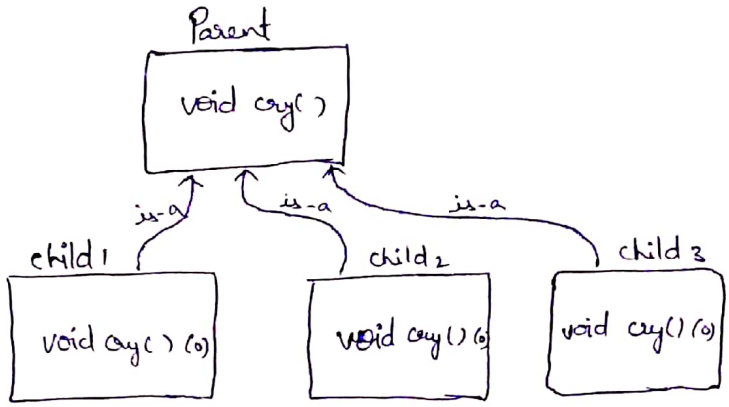
```
void sleep()
{
  System.out.println("Sleeps at night");
}
void breathe()
{
  System.out.println("Breathes oxygen");
}
}

class Tiger extends Animal
{
  void eat()
  {
    System.out.println("Hunts and eat");
  }
  void run()
  {
    System.out.println("Tiger runs fast");
  }
}
class Deer extends Animal
{
  void eat()
  {
    System.out.println("graces and eat");
  }
  void leap()
  {
    System.out.println("Deer leaps");
  }
}
```

```
class Monkey extends Animal
{
  void eat()
  {
    System.out.println("Steals and eat");
  }
  void climbs()
  {
    System.out.println("Monkey jumps");
  }
}

class LaunchAnimal
{
  public static void main(String args[])
  {
    Tiger t=new Tiger();
    Deer d=new Deer();
    Monkey m=new Monkey();
     t.eat();
     t.sleep();
     t.breathe();
     t.run();
    System.out.println("*****");
     d.eat();
     d.sleep();
     d.breathe();
     d.leaps();
    System.out.println("*****");
      m.eat();
      m.sleep();
       m.breathe();
        m.climbs();
     System.out.println("*****");
  }
}
```

O/p:-

```
Hunts and eat
Sleeps at night
Breathes oxygen
Tiger runs fast
*****
Grazes and eat
Sleeps at night
Breathes oxygen
Deer leaps
*****
Steals and eat
Sleeps at night
Breathes oxygen
Monkey jumps
*****
```

18/7/19



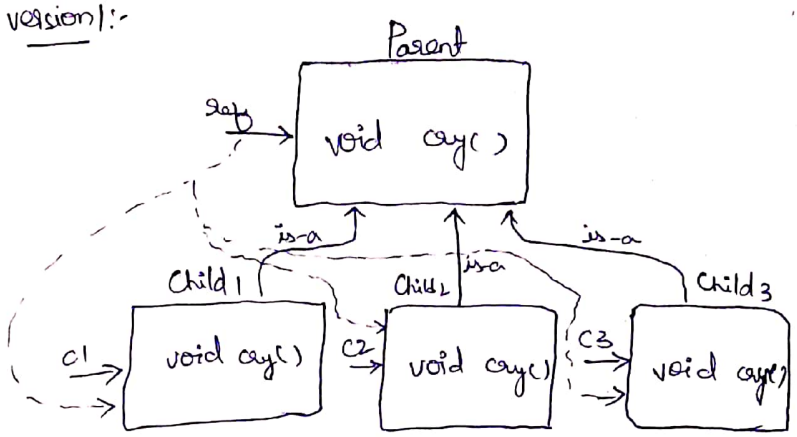

Non- Polymorphism version:

```
class Parent
 {
    void cry()
    {
      System.out.println("Parent is crying");
    }
 }
class Child1 extends Parent
{
   void cry()
   {
     System.out.println("Child1 is crying in low voice");
   }
}
class Child2 extends Parent
{
  void cry()
  {
    System.out.println("Child2 is crying in medium voice");
  }
}
class Child3 extends Parent
{
   void cry()
   {
     System.out.println("Child3 is crying in high voice");
   }
}
```

```
class LaunchParent
{
  public static void main(String args[])
  {
    Child1  c1=new Child1();
    Child2  c2=new Child2();
    Child3  c3=new Child3();

    c1.cry();
    c2.cry();
    c3.cry();
  }
}
```

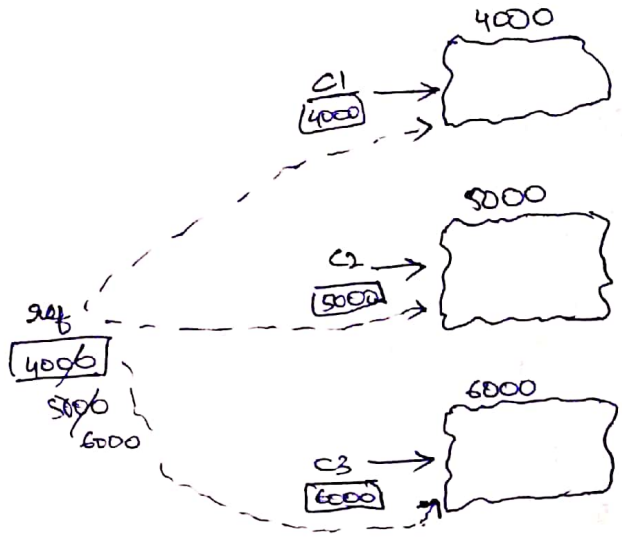
O/P:- Child1 is crying in low voice
Child2 is crying in medium voice
Child3 is crying in high voice.

Polymorphism version1:-




```
class Parent
{
  void say()
   {
    System.out.println("Parent is saying");
   }
}
class Child1 extends Parent
{
  void say()
   {
    System.out.println("Child1 cries at low voice");
   }
}
class Child2 extends Parent
{
  void say()
   {
    System.out.println("Child2 cries at medium voice");
   }
}
class Child3 extends Parent
{
  void say()
   {
    System.out.println("Child3 cries at high voice");
}}
```

```
class launchParent
{
  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();
  }
}
```

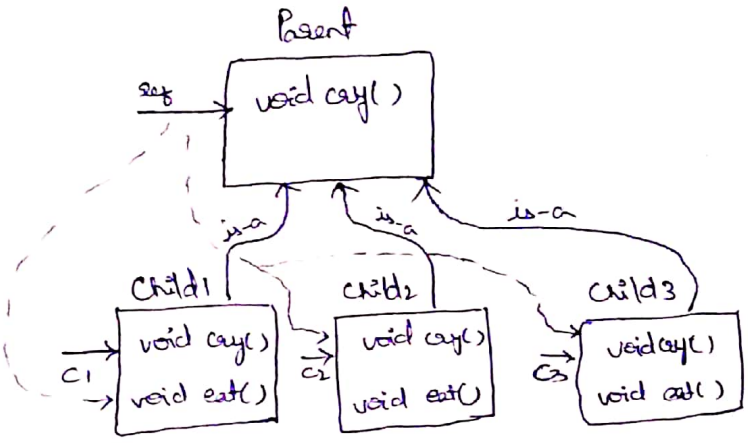


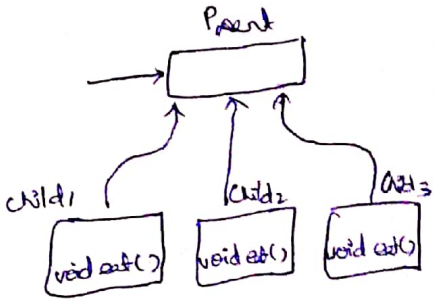

O/p:- Child1 cries at low voice
Child2 cries at medium voice
Child3 cries at high voice

NOTE:

(1) In the above program, even though we have achieved polymorphism code compactness has not been implemented.

19/7/19



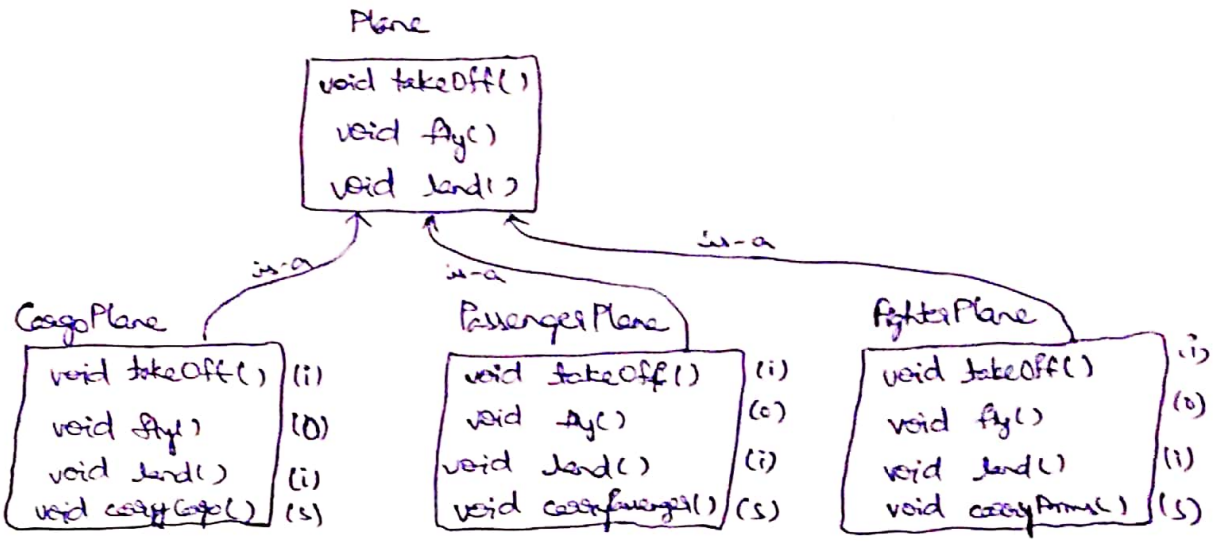

```
class Parent
{
  void cay()
   {
    System.out.println("Parent is caying");
   }
}
class Child1 extends Parent
{
   void cay()
   {
     System.out.println("Child1 is caying");
   }
   void eat()
   {
     System.out.println("Child1 is eating less food");
   }
}
class Child2 extends Paren
{
   void cay()
   {
     System.out.println("Child2 is caying");
   }
```

```
    void eat()
    {
      System.out.println("Child2 eats sufficient food");
    }
}

class Child3 extends Parent
{
  void cry()
    {
      System.out.println("Child3 is crying");
    }
    void eat()
    {
      System.out.println("Child3 eats more than sufficient food");
    }
}
class LaunchParent
{
  public static void main(String args[])
  {
     Child1 c1=new Child1();
     Child2 c2=new Child2();
     Child3 c3=new Child3();
     Parent ref;
      ref=c1;
      ref.cry();
     ((Child1)(ref)).eat(); ----> down casting or explicit typecasting
      ref=c2;
      ref.cry();
     ((Child2)(ref)).eat();
      ref=c3;
      ref.cry();
     ((Child3)(ref)).eat();
}}
```

NOTE:

→ Using parent type reference we cannot access specialised methods directly.

→ In order to access the specialised method of child class using parent reference we have to program perform down casting.

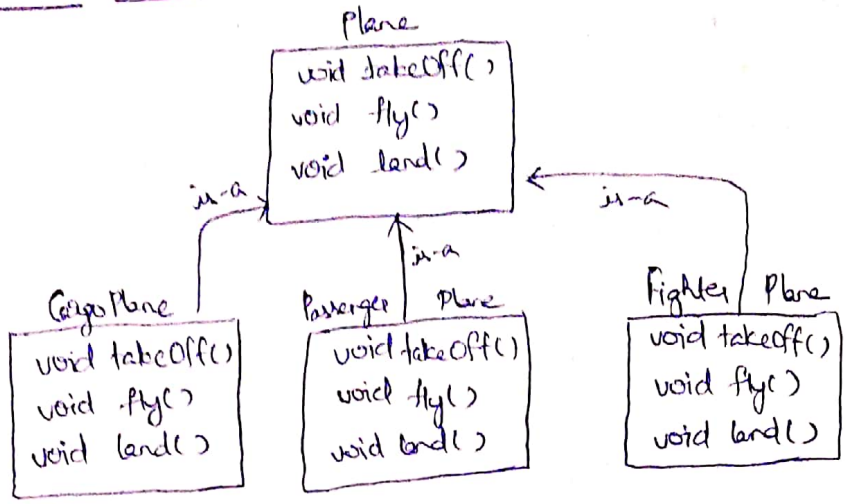



```
class Plane
{
  void takeOff()
  {
    System.out.println("Plane is taking off");
  }
  void fly()
  {
    System.out.println("Plane is flying");
  }
  void land()
  {
    System.out.println("Plane is landing");
  }
}
```

```
class CargoPlane extends Plane
{
  void fly()
  {
    System.out.println("Plane is flying at lower heights");
  }
  void carryCargo()
  {
    System.out.println("Plane carries cargo");
  }
}
class PassengerPlane extends Plane
{
  void fly()
  {
    System.out.println("Plane is flying at medium heights");
  }
  void carryPassenger()
  {
    System.out.println("Plane carries passengers");
  }
}
class FighterPlane extends Plane
{
  void fly()
  {
    System.out.println("Plane is flying at lower heights");
  }
  void carryArms()
  {
    System.out.println("Plane carries arms");
  }
}
```

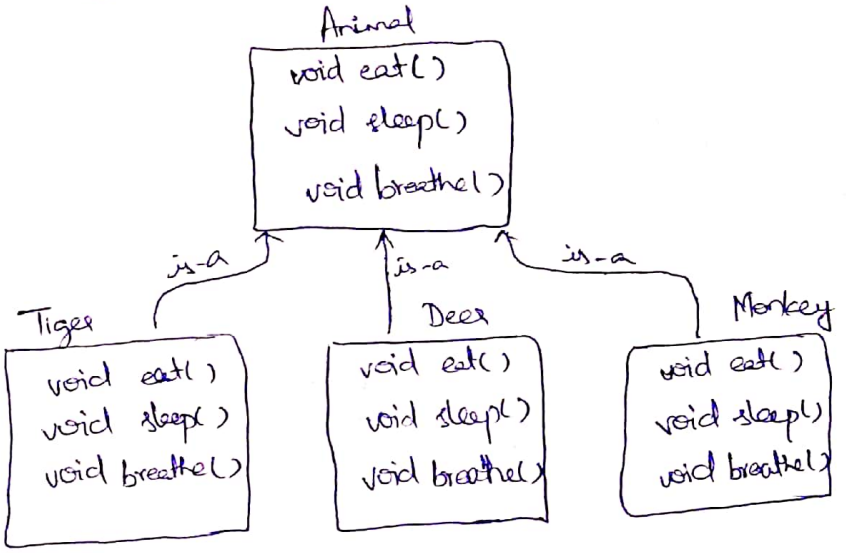
```
class LaunchPlane
{
  public static void main(String args[])
  {
    CargoPlane cp=new CargoPlane();
    PassengerPlane pp=new PassengerPlane();
    FighterPlane fp=new FighterPlane();

    Parent ref;
    ref=cp;
    ref.takeOff();
    ref.fly();
    ref.land();
    ((CargoPlane)(ref)).carryCargo();
    ref=pp;
    ref.takeOff();
    ref.fly();
    ref.land();
    ((PassengerPlane)(ref)).carryPassengers();
    ref=fp;
    ref.takeOff();
    ref.fly()
    ref.land();

    ((FighterPlane)(ref)).carryArms();
  }
}
```

Polymorphism version2 :-




```
class Airport
{ 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);
  }
}
```

Polymorphism version 2:-




```
class Jungle
{
  void permit(Animal ref)
  {
    ref.eat();
    ref.sleep();
    ref.breathe();
  }
}
class LaunchAnimal
{
  public static void main(String args[])
  {
    Tiger t=new Tiger();
    Deer d=new Deer();
    Monkey m=new Monkey();
    Jungle j=new Jungle();
    j.permit(t);
    j.permit(d);
    j.permit(m);
  }
}
```

22/7/19

## Visibility of access specifiers:

Package 1

```
class Demo1
{
  public int a;
  protected int b;
  int c;
  private int d;
  void disp()
  {
    =
  }
}
```

```
class Demo2 extends Demo1
{
  void disp()
  {
    =
  }
}
```

```
class Demo3
{
  void disp()
  {
    =
  }
}
```

Package 2

```
class Demo4 extends Demo1
{
  void disp()
  {
    =
  }
}
```

```
class Demo5
{
  void disp()
  {
    =
  }
}
```

| | Demo1 | Demo2 | Demo3 | Demo4 | Demo5 |
|---|---|---|---|---|---|
| public | ✓ | ✓ | ✓ | ✓ | ✓ |
| protected | ✓ | ✓ | × | ✓ | × |
| default/ No access specifier | ✓ | ✓ | ✓ | × | × |
| private | ✓ | × | × | × | × |

✓ – accessible

× – not accessible

## Rules of overriden methods:

(1) We cannot reduce the visibility of overriden methods.

Demo1

```
class Demo
{
  public void disp()
  {
    =
  }
}
```

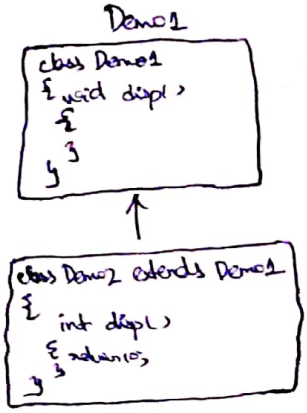
↑

```
class Demo2 extends Demo1
{
  void disp()
  {
  }
}
```

```
class Demo1
{
  public void disp()
  {
  }
}
class Demo2 extends Demo1
{
  void disp()
  {
  }
}
class Demo
{
  public static void main(String args[])
  {
    Demo1 d2=new Demo2();
    d2.disp();
  }
}
```

O/p:- Error

2) The return type of child class should be same as that of parent class.

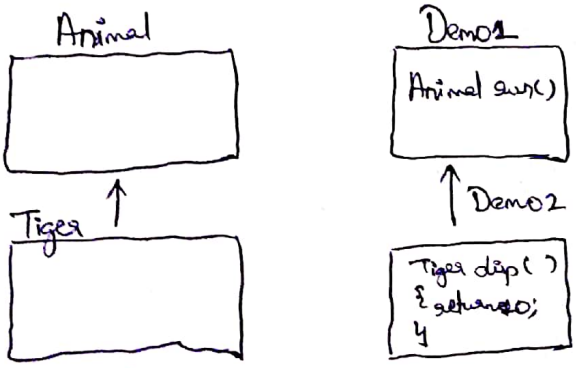


```
class Demo1
{
  void disp()
  {
  }
}

class Demo2 extends Demo01
{
  int disp()
  {
    return 10;
  }
}
class Demo
{
  public static void main(String args[])
  {
    Demo2 d2=new Demo2();
    d2.disp();
  }
}
```

O/p: Error

23/7/19 → [Co-variant return types]

Rule (3): If a method in child class and a method in parent class have different return types it is still accepted if there is "is-a" relationship between return types



```
class Animal
{

}
class Tiger extends Animal
{

}
class Demo1
{
 Animal disp()
 {
  Animal a=new Animal();
  return a;
 }
}
class Demo2 extends Demo1
{
  Tiger disp()
  {
    Tiger t=new Tiger();
    return t;
  }
}
class LaunchDemo2
{
  public static void main(String args[])
  {
    Demo2 d2=new Demo2();
    d2.disp();
  }
}
```

## Delegation Models:-

Delegation model is an alternative to inheritance. Sometimes inheritance may not be a good approach in programs and in such cases we have to make use of delegation model.

Using inheritance:-

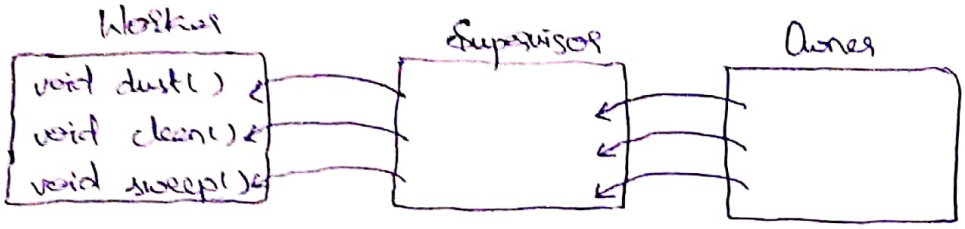

```
class Worker
{
  void dust()
  {
   System.out.println("dusting");
  }
  void sweep()
  {
   System.out.println("sweeping");
  }
  void clean()
  {
   System.out.println("cleaning");
  }
}
class Owner
{
 public static void main(String args[])
 {
   Supervisor s=new Supervisor();
   s.dust();
   s.sweep();
   s.clean();
  }
}
```

O/p:- Dusting
sweeping
cleaning

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

```
class Supervisor extends Worker
{

}
```

Using delegation model:




```
class Worker
{
 void dust()
 {
 System.out.println("dusting");
 }
 void sweep()
 {
 System.out.println("sweeping");
 }
 void clean()
 {
  System.out.println("cleaning");
 }
}
```

```
class Supervisor
{
 Worker w=new Worker();
 Void dust()
 {
  w.dust();
 }
 void sweep()
 {
  w.sweep();
 }
 void clean()
 {
  w.clean();
 }
}
```

```
class Owner
{
public static void main(String
                        args[])
{
Supervisor S=new Supervisor();
 s.dust();
 s.sweep();
 s.clean();
 }
}
```

O/p: dusting
sweeping
cleaning

24/7/19

Usage of final keyword:

```
final class Demo1
{
  void disp()
  {
  }
}
class Demo2 extends Demo1
{
  void disp()
  {
  }
}
class LaunchDemo2
{
  public static void main(String args[])
  {
    Demo2 d2=new Demo2();
    d2.disp();
  }
}
```

O/p:- Error: Cannot inherit from final Demo1

→ If final keyword is used on class then none of other classes will be able to inherit from final class.

```
class Demo1
{
  final void disp()
  {
    System.out.println("inside Demo1");
  }
}
class Demo2 extends Demo1
{
  void disp()
  {
    System.out.println("inside Demo2");
  }
}
class LaunchDemo2
{
  public static void main(String args[])
  {
    Demo2 d2=new Demo2();
    d2.disp();
  }
}
```

O/p: Error! cannot override disp() in Demo2 from disp() in Demo1 which is final.

→ Here from Demo1 can be inherited but method cannot be overriden.

```
class Demo1
{
  void disp()
  {
   final int a=10;
   System.out.println(a);
   a=20;
  }
}

class LaunchDemo1
{
  public static void main(String args[])
  {
   Demo1 d1=new Demo1();
   d1.disp();
  }
}
```

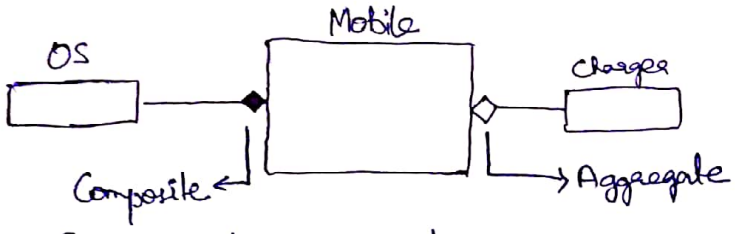
O/p: error: cannot assign a value to final variable a

→ If final keyword is applied for variable no body is able to change the value of variable or, in other words it will behave like constant.

## Aggregation and Composition:-

→ Composition refers to tight coupling where composite object will be accessible as long as enclosing object is accessible.

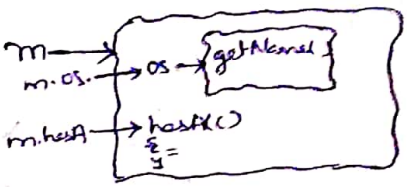
→ Aggregation refers to loose coupling where aggregate object will be accessible even after the enclosing object is not accessible.

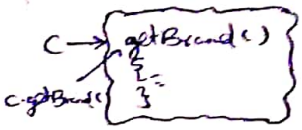



First consider composite classes then aggregate classes and next enclosing classes.

```
class OS
{
  String name;
  Os(String name)
  {
    this.name = name;
  }
  String getName()
  {
    return name;
  }
}
class Charger
{
  String brand;
  Charger(String brand)
  {
    this.brand = brand;
  }
  String getBrand()
  {
    return brand;
  }
}
```

```
class Mobile
{
  Os os=new Os("pie");
  void hasA(Charger c)
  {
    System.out.println(c.getBrand());
  }
}
class LaunchMobile
{
  public static void main(String args[])
  {
    Mobile m=new Mobile();
    Charger c=new Charger("SAMSUNG");
    System.out.println(m.os.getName());
    System.out.println(c.getBrand());
    m.hasA(c);
    m=null;
    System.out.println(m.os.getName());
    System.out.println(c.getBrand());
    m.hasA(c);
  }
}
```

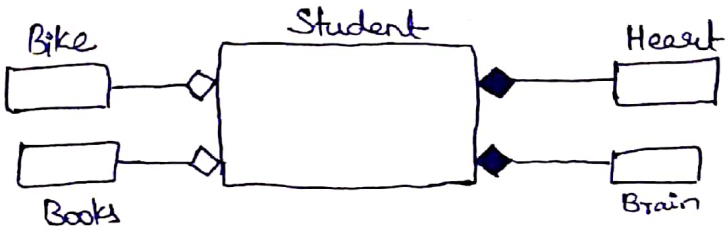



O/P:- Pie
SAMSUNG
SAMSUNG

Line 19: Error: NullPointerException

25/7/19

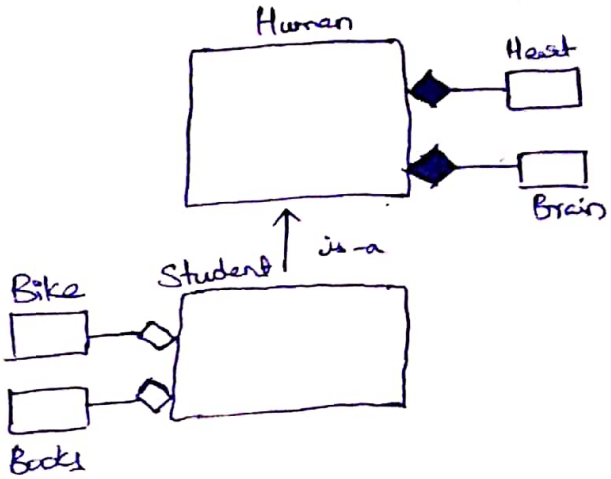



```
class Heart
{
  int pulserate;
  Heart(int pulserate)
  {
    this.pulserate = pulserate
  }
  int getPulseRate()
  {
    return pulserate;
  }
}
class Brain
{
  int iq;
  Brain(int iq)
  {
    this.iq=iq;
  }
  int getIq()
  {
    return iq;
  }
}
```

```
class Bike
{
  int mileage;

  Bike(int mileage)
  {
    this.mileage = mileage;
  }
  int getMileage()
  {
    return mileage;
  }
}

class Books
{
  int pg;

  Books(int pg)
  {
    this.pg=pg;
  }

  int getPg()
  {
    return pg;
  }
}

class Student
{
  Heart h=new Heart(70);
  Brain b=new Brain(75);
  void hasA(Bike bi)
  {
   System.out.println(bi.getMileage);
  }
```

```
void hasA(Books bo)
{
  System.out.println(bo.getPg());
}

class LaunchStudent
{
  public static void main(String args[])
  {
    Student s=new Student();
    Bike bi=new Bike(50);
    Books bo=new Books(200);
    System.out.println(s.h.getPulseRate());
    System.out.println(s.b.getIq());
    System.out.println(bi.getMileage());
    System.out.println(bo.getPg());
    s.hasA(bi);
    s.hasA(bo);
     s=null;
    System.out.println(s.h.getPulseRate());
    System.out.println(s.b.getIq());
    System.out.println(bi.getMileage());
    System.out.println(bo.getPg());
    s.hasA(bi);
    s.hasA(bo);
  }
}
```

O/p: 70
75
50
200
50
200
Exception: NullPointerException




```
class Heart
{
 int pulserate
  Heart(int pulserate)
  {
   this.pulserate=pulserate;
  }
  int getPulseRate()
  {
    return pulserate;
  }
}
```

```
class Brain
{
  int iq;

  Brain(int iq)
  {
    this.iq=iq;
  }
   int getIq()
   {
    return iq;
   }
}
```

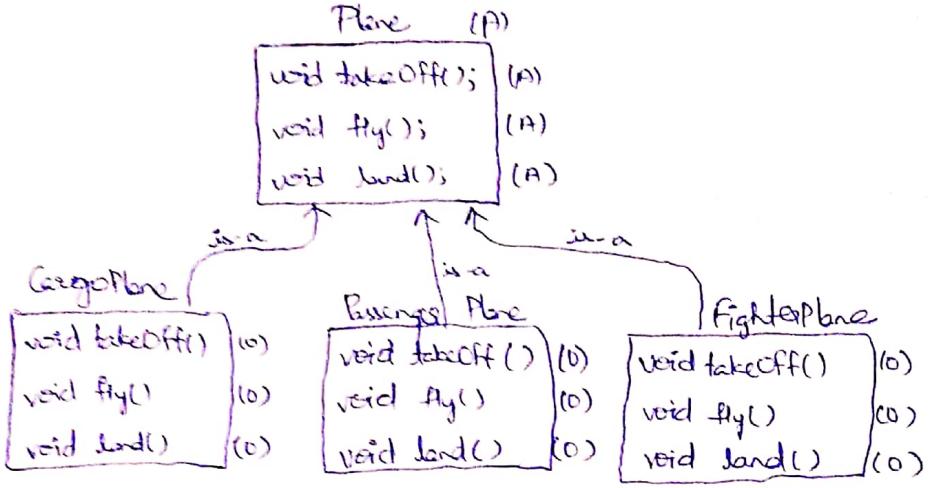
```
class Bike
{
  int mileage;
  Bike(int mileage)
  {
    this.mileage=mileage;
  }
  int getMileage()
  {
    return mileage;
  }
}
```

```
class Books
{
  int pg;
  Books(int pg)
  {
    this.pg=pg;
  }
int getPg()
  {
    return pg;
  }
}
```

```
class Human
{
  Heart h=new Heart(70);
  Brain b=new Brain(75);
}

class Student extends Human
{
  void hasA(Bike bi)
  {
    System.out.println(bi.getMileage());
  }
  void hasA(Books bo)
  {
    System.out.println(bo.getPython());
  }
}

class LaunchStudent
{
  public static void main(String args[])
  {
    Student s=new Student();
    Bike bi=new Bike(50);
    Books bo=new Books(200);

    System.out.println(s.h.getPulseRate());
    System.out.println(s.b.getIq());
    System.out.println(bi.getMileage());
    System.out.println(bo.getPg());
    s.hasA(bi);
    s.hasA(bo);
    s=null;
    System.out.println(s.h.getPulseRate());
    System.out.println(s.b.getIq());
    System.out.println(bi.getMileage());
    System.out.println(bo.getPg());
    s.hasA(bi);
    s.hasA(bo);
  }
}
```

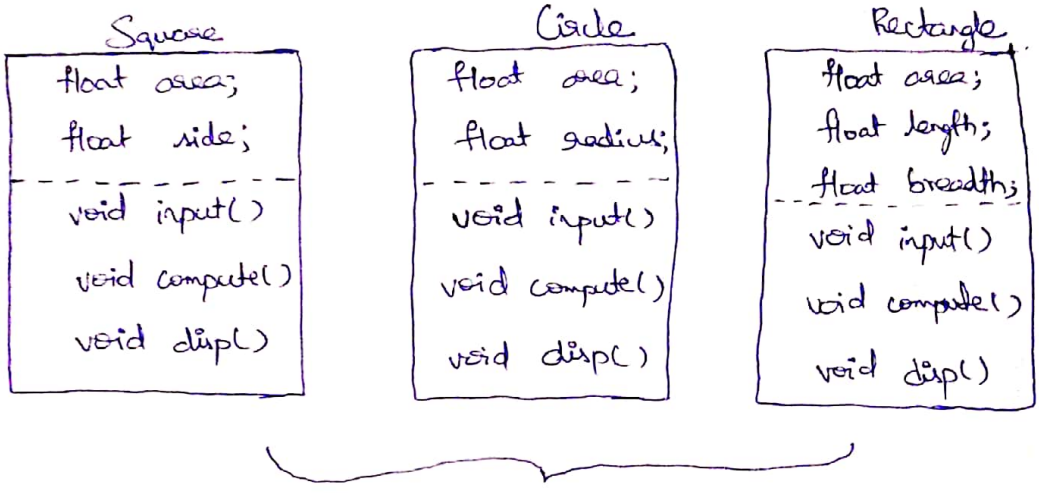
26/7/19

## Abstract classes:

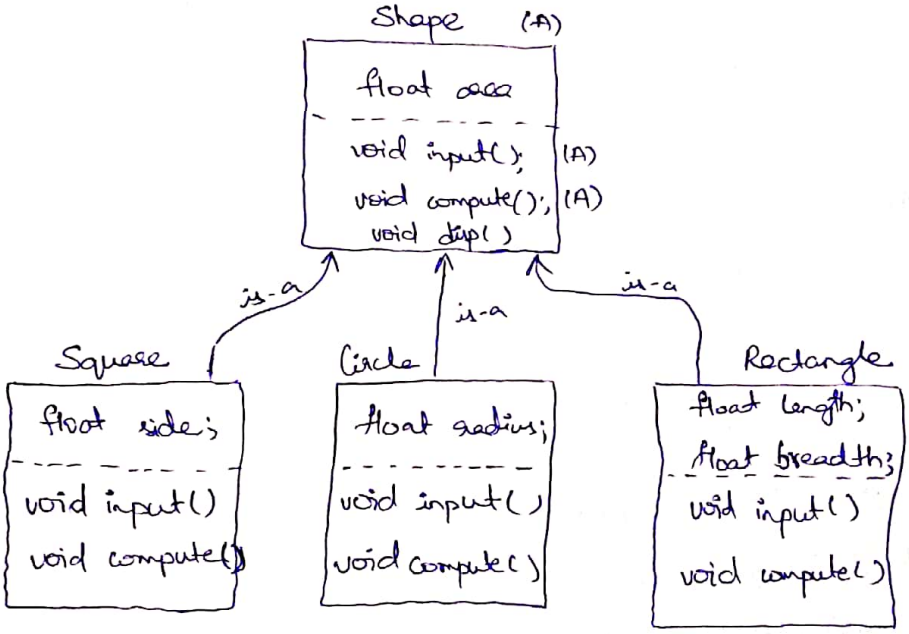



```
abstract class Plane
{
  abstract void takeOff();
  abstract void fly();
  abstract void land();
}
class CargoPlane extends Plane
{
  void takeOff()
  {
    System.out.println("cargoplane requires longer runway to takeoff");
  }
  void fly()
  {
    System.out.println("cargo plane flies at lower heights");
  }
  void land()
  {
    System.out.println("cargoplane requires longer runway to land");
  }
}
```

```
class PassengerPlane extends Plane
{
  void takeOff()
  {
    System.out.println("Passenger plane requires medium runway to take off");
  }
  void fly()
  {
    System.out.println("Passenger plane flies at medium heights");
  }
  void land()
  {
    System.out.println("Passenger plane requires medium runway to land");
  }
}
class FighterPlane extends Plane
{
  void takeOff()
  {
    System.out.println("Fighter plane requires shorter runway to takeOff");
  }
  void fly()
  {
    System.out.println("Fighter plane flies at higher heights");
  }
  void land()
  {
    System.out.println("Fighter plane requires shorter runway to land");
  }
}

class Airport extends Plane
{
  void permit(plane ref)
  {
    ref.takeOff();
    ref.fly();
    ref.land();
  }
}
  class LaunchPlane
  {
    public static void main(String args[])
    {
      Airport a=new Airport();
      CargoPlane cp=new CargoPlane();
      PassengerPlane pp=new PassengerPlane();
      FighterPlane fp=new FighterPlane();
        a.permit(cp);
        a.permit(pp);
        a.permit(fp);
}}
```




"Stupid Architecture".

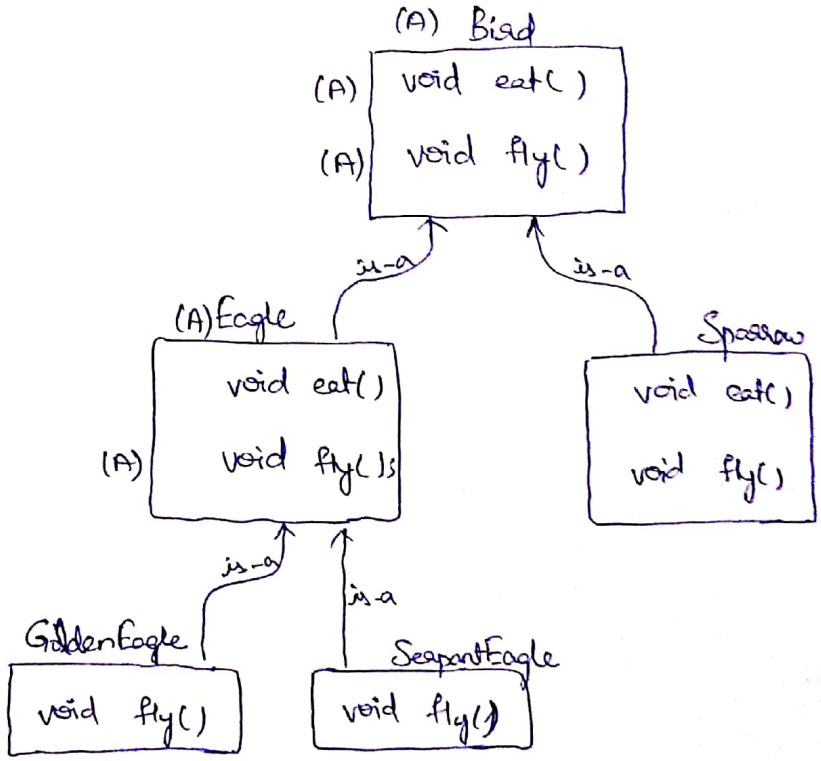



```
import java.util.Scanner;
Abstract class Shapee
{
  float area;

  abstract void input();
  abstract void compute();
   void disp()
   {
     System.out.println(area);
   }
}
```

```
class Square extends Shapee
{
 float side;
  void input()
  {
    Scanner scan=new Scanner(System.in);
    System.out.println("Enter the side");
    side= scan.nextFloat();
  }
 void compute()
 {
   area= side * side;
  }
}
class Circle extends Shapee
{
 float radius;
  void input()
  {
    Scanner scan=new Scanner(System.in);
    System.out.println("enter the radius");
    radius = scan.nextFloat();
  }
  void compute()
  {
    area= 3.142 * radius * radius;
  }
}
 class Geometry
 {
   void permit(Shape ref)
   {
     ref.input();
     ref.compute();
     ref.disp();
   }
}
```

```
class LaunchShapes
{
  public static void main(String args[])
  {
    Geometry g=new Geometry();
    Square s=new Square();
    Circle c=new Circle();
    Rectangle r=new Rectangle();
    g.permit(c);
    g.permit(s);
    g.permit(r);
  }
}
```

27/7/19




```
import java.util.Scanner;
abstract class Bird
  {
     abstract void eat();
     abstract void fly();
  }
```

```
abstract class Eagle extends Bird
{
  void eat()
  {
   System.out.println("Eagle is eating");
  }
 abstract void fly();
}

class GoldenEagle extends Eagle
{
  void fly()
  {
   System.out.println("Golden Eagle flies over oceans");
  }
}

class SeaportEagle extends Eagle
{
  void fly()
  {
   System.out.println("SeaportEagle flies over mountains");
  }
}

class Sparrow extends Bird
{
  void eat()
  {
   System.out.println("Sparrow is eating");
  }
  void fly()
  {
   System.out.println("Sparrow flies at lower heights");
  }
}
```

```
class BirdType Sky
{
  void permit(Bird obj)
  {
    obj.eat();
    obj.fly();
  }
}

class LaunchBird
{
  public static void main(String args[])
  {
    Sky s=new Sky();
    Sparrow sp=new Sparrow();
    GoldenEagle ge=new GoldenEagle();
    SeapartEagle se=new SeapartEagle();
    s.permit(sp);
    System.out.println("#####");
    s.permit(ge);
    System.out.println("#####");
    s.permit(se);
    System.out.println("#####");
  }
}
```

o/p:

NOTE:-

(1) Abstract methods are such methods for which body doesnot exist.

(2) If a method in class is abstract then we have to declare class also as abstract.

(3) We can have abstract methods and abstract classes but not abstract variables.

NOTE:

(1) Abstract class can have all the methods as abstract.

(2) An abstract class can have few methods as abstract and few methods as concrete.

(3) A concrete class cannot have any abstract methods.

(4) We cannot create any object of abstract class (abstract class cannot be instantiated)

Ex:

```
abstract class Bird
{
   abstract void fly();
}
class LaunchBird
{
   public static void main(String args[])
   {
     Bird b=new Bird();
   }
}
```

O/p:- Error

(5)
```
abstract class Demo
{
  static void disp()
  {
    System.out.println("disp");
  }
}
class launchDemo
{
  public static void main(String args[])
  {
    Demo.disp();
  }
}
```

O/p: disp

Abstract class can contain all methods which are concrete.

30/7/19

→ Even if one method is class

→ An abstract class and abstract method cannot be final.
An abstract class cannot be final because no class would be able to inherit abstract class.

Abstract method cannot be final because no body able to override it, no body give body to abstract method.

```
abstract class Demo1
{
  final abstract void disp();
}
class Demo2 extends Demo1
{
  void disp()
  {
  }
}
```

```
class LaunchDemo
{
  public static void main(String args[])
  {
    Demo2 d2=new Demo2();

  }
}
```

O/p: error: illegal combination of identifiers : abstract and final

error: disp() in Demo2 cannot override disp() in Demo1

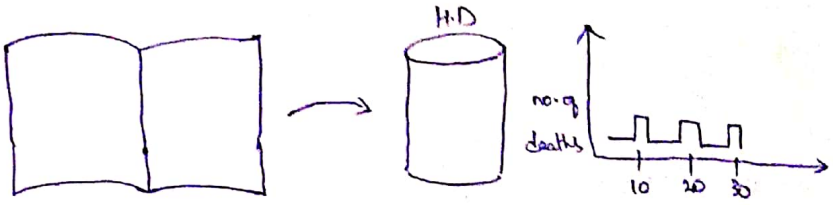
NOTE:

(1) Constructors cannot be abstract because in every constructor we would compulsory will have either this method or, super method

(2) Abstract classes could contain constructor because when we create the object for child class we would call child class constructor and inside child class constructor we will have super method call which will take the control to the parent class constructor.

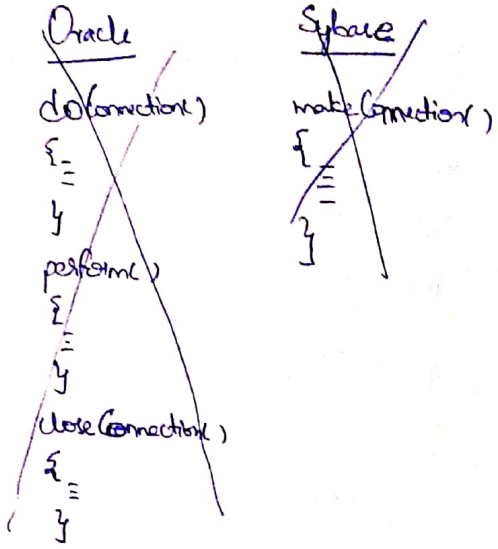
# INTERFACE:-

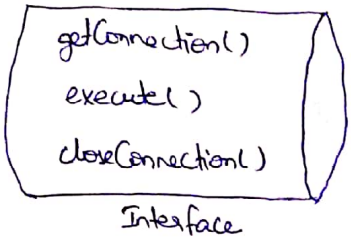
1920 – Ledgers
1930 – "
1940 – "
1950 – "
1960 – "
1970 – "
1980 – Big organisations
1990 –




DBMS

Oracle 20% 60%
(50%) Sybase 20% 10% → Java
(1yr+1yr) Informix 20% 10%
(1+1) DB2 20% 10%
MSExcel 20% 10%

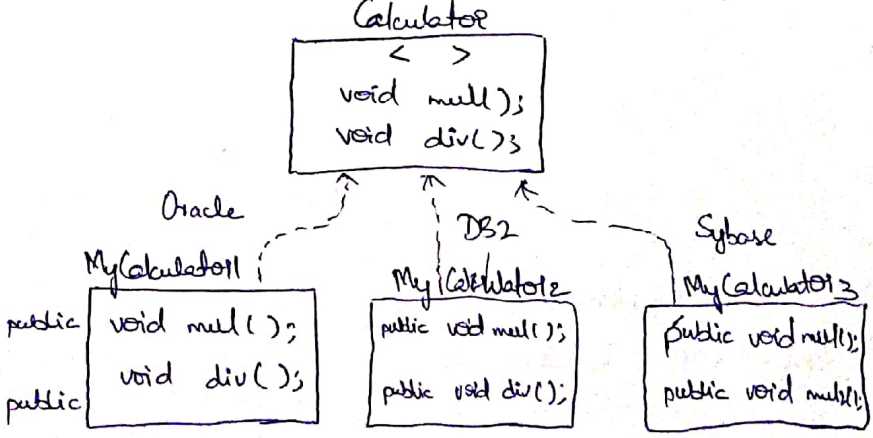



→ An interface refers to collection of public abstract methods

Rule 1:-

Interface can have any number of implementations.



```
import java.util.Scanner;

interface Calculator
{
   void mul();
   void div();
}

class MyCalculator1 implements Calculator
{
   int a=20;
   int b=10;
   int c;

   public void mul()
   {
      c=a*b
      System.out.println("Multiplication="+c);
   }
   public void div()
   {
      c=a/b;
      System.out.println("Division="+c);
   }
}

class MyCalculator2 implements Calculator
{
   public void mul()
   {
      int a;
      int b;
      Scanner scan=new Scanner(System.in);
      System.out.println("Enter a & b values:");
      a= scan.nextInt();
      b= scan.nextInt();
```

```
int c;
 c=a*b;
 System.out.println("Multiplication:"+c);
}

public void div()
{
 int a;
 int b;
 Scanner scan=new Scanner(System.in);
 System.out.println("Enter a & b value:");
 a=scan.nextInt();
 b=scan.nextInt();
 int c;
 c=a/b;
 System.out.println("Division="+c);
 }
}

class MyCalculator3 implements Calculator
{
  public void mult()
  {
    int a;
    int b;
    Scanner scan=new Scanner(System.in);
    System.out.println("Enter a & b value:");
    a=scan.nextInt();
    b=scan.nextInt();
    int c;
    if(a==0 || b==0)
    {
      System.out.println("Invalid inputs");
    }
```

```
    else
    {
      c=a*b;
      System.out.println("Multiplication = "+c);
    }
}

public void div()
{
  int a;
  int b;
  Scanner scan=new Scanner(System.in);
  System.out.println("Enter a & b values");
  a= scan.nextInt();
  b= scan.nextInt();
  int c;
  if(b==0)
  {
    System.out.println("Invalid Inputs");
  }
  else
  {
     c=a/b;
     System.out.println("Division"+c);
  }
}
}
class LaunchCalculator
{
  public static void main(String args[])
  {
    MyCalculator1 mc1=new MyCalculator1();
    MyCalculator2 mc2=new MyCalculator2();
    MyCalculator3 mc3=new MyCalculator3();
```

```
        mc1.mul();
        mc2.div();

        mc1.mul();
        mc2.div();

        mc3.mul();
        mc3.mul();
    }
}
```

O/P:- Multiplication = 200

Division = 2

Enter a & b value

15

2

30

Enter a & b value

30

2

Division=15

Enter a & b value

20

5

Multiplication=100

Enter a & b value

50

5

Division=10

31/7/19 Polymorphic version:

```
interface class Calculator
{ void mul();
  void div();
}
class Maths
{ void permit(Calculator calc)
  { calc.mul();
    calc.div();
  }
}
class MyCalculator1 implements Calculator
{
  public void mul()
  { int a=20,b=10, c;
    c=a*b;
    System.out.println(c);
  }
  public void div()
  { int a=20, b=10, c;
      c=a/b;
      System.out.println(c);
  }
}
class MyCalculator2 implements Calculator
{ =
}
class MyCalculator3 implements Calculator
{ =
}
class LaunchCalculator
{ public static void main(String args[])
  { Maths m=new Maths();
    MyCalculator1 mc1=new MyCalculator1();
```

31/7/19

Rule2: Interface promotes polymorphism

Rule3: Interface can achieve standardization

Rule4: We can't create object of an interface.

Ex:
```
interface Calculator
{
  void null();
}
class MyCalculator1
{
  public static void main(String args[])
  {
    Calculator calc=new Calculator();
  }
}
```

O/p: Error: abstract cannot be instantiated.

Rule 5:

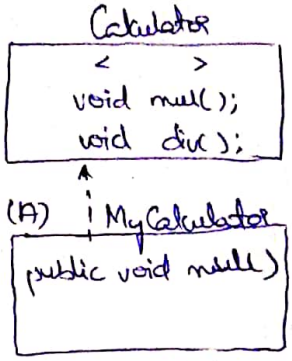
Even though the object of interface cannot be created, reference of interface can be created.

```
interface Calculator
{
  void null();
}
class MyCalculator
{
  public static void main(String args[])
  {
    Calculator calc;
  }
}
```

Rule 6:-

It is not mandatory for implementing class to give body for all method in interface, However if an implementing class is not giving a body for all methods then we should declare class as abstract

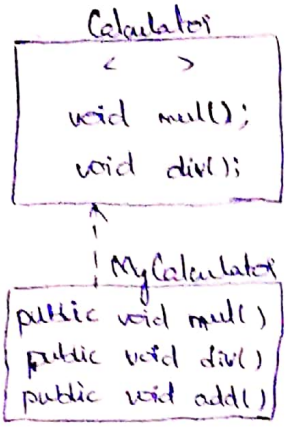
```
interface Calculator
{
  void mul();
  void div();
}

abstract class MyCalculator implements Calculator
{
  public void mul()
  {
    int a=10;
    int b=20;
    int c=a*b;
    System.out.println(c);
  }
}
```




Rule If there is no access specifier for a method present inside a class then it is treated as default and if there is a method inside a interface without access specifier then it is treated as public.

Rule 7:-

An implementing class need not have only implemented methods in addition to that they can also have specialized methods, however using the interface reference we may not be able to access the specialized methods directly they should be accessed through down casting.




```
byte a=120;
double b;
  b=a;
```

Implicit Typecasting / Upcasting

```
double a=124.12
byte b=5
b=(byte)a
```

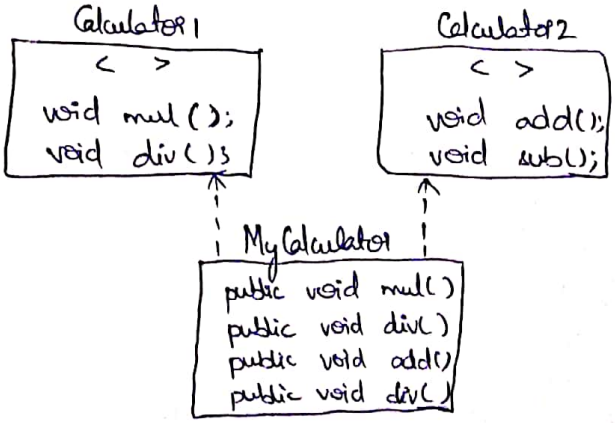
Explicit typecasting / downcasting

```
interface Calculator
{
  void mul();
  void div();
}

class MyCalculator implements Calculator
{
   public void mul()
   {
     int a=20;
     int b=10;
     int c;
      c=a*b;
     System.out.println(c);
   }

   public void div()
   {
     int a=20;
     int b=10;
      int c;
      c=a/b;
      System.out.println(c);
   }
```

```
public void add()
{
    int a=10;
    int b=20;
    int c=a+b;
    System.out.println(c);
}

class LaunchCalculator
{
  public static void main(String args[])
  {
     Calculator calc=new MyCalculator();
      calc.mul();
      calc.div();
     ((MyCalculator)(Calc)).add();
  }
}
```

Rule 8:

A class can implement multiple interfaces because in this case there is not any diamond shape problem.



```
interface Calculator1
{
  void mul();
  void div();
}
interface Calculator2
{
  void sub();
  void add();
}

class MyCalculator implements Calculator1, Calculator2
{
  public void mul()
  {
    int a=20;
    int b=10;
    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);
  }
  public void add();
  {
    int a=20;
    int b=10;
    int c=a+b;
    System.out.println(c);
  }
```

```
public void sub()
{
  int a=20;
  int b=10;
  int c=a-b;
  System.out.println(c);
 }
}

class LaunchCalculator
 {
   public static void main(String args[])
   {
    MyCalculator calc=new MyCalculator();
    calc.mul();
    calc.div();
    calc.add();
    calc.sub();
   }
 }
```

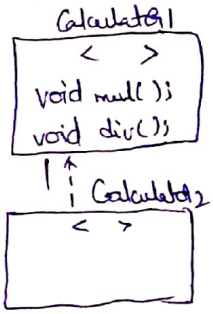
Rule 9:

Difference between abstract class and interface:-

| | Abstract | Interface |
|---|---|---|
| 1) | 'abstract' keyword is used to create abstract classes. | 'interface' keyword is used to create interface |
| 2) | abstract class cannot extend multiple abstract classes. | It can have only abstract method. |
| 3) | Abstract class can have constructors. | Interface cannot have constructors. |
| 4) | By default methods may not be public in abstract classes. | By default, interface methods are public |
| 5) | Abstract classes can have main method. | Interface cannot have main method. |

Rule 9:

One interface cannot implements another interface.




```
interface Calculator1
{
  void mul();
  void div();
}
interface Calculator2 implements Calculator1
{
  =
 }
```

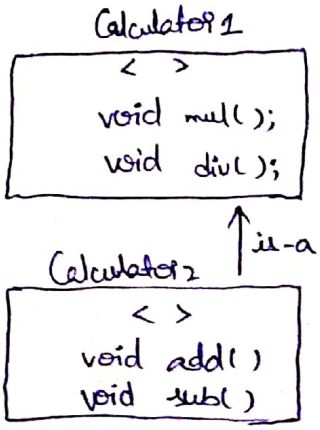
O/p:- Error: interface Calculator1 '{' expected

Calculator2 cannot implement Calculator1

1/8/19

Rule 10 :-

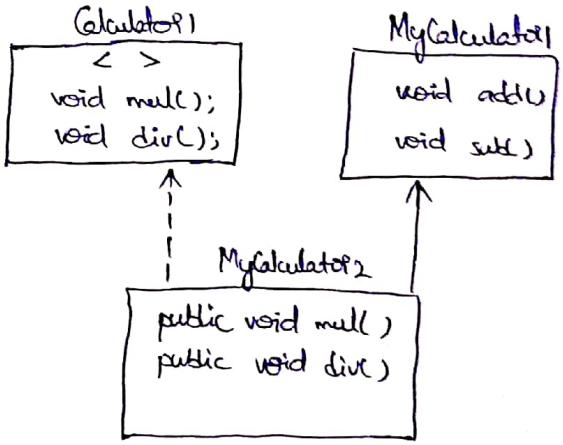
One interface can extend another can extend another interface.



```
interface Calculator1
{
  void mul();
  void div();
}
interface Calculator extends Calculator1
 {
  void add();
  void sub();
}
class LaunchCalculator
 {
   public static void main(String args[])
   {
   }
 }
```

Rule11:

A class can extend another class and also can implement an interface




First extend class then implement interface

```
interface Calculator1
{
  void mul();
  void div();
}
class MyCalculator1
 {
   void add()
    {
```

```
        int a=10;
        int b=20;
        int c=a+b;
        System.out.println(c);
    }

    void sub()
    {
        int a=20;
        int b=10;
        int c= a-b;
        System.out.println(c);
    }
}

class MyCalculator2 extends MyCalculator1 implements Calculator1
{
    public void mul()
    {
        int a=20;
        int b=10;
        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 LaunchCalculator
{
  public static void main(String args[])
  {
    MyCalculator2  mc2=new MyCalculator2();
     mc2.add();
     mc2.sub();
     mc2.mul();
     mc2.div();
  }
}
```

Rule 12:-

All the methods inside a interface are automatically public and abstract. The methods are public because it should be accessible to all implementing classes and the methods are abstract because there is no body for methods.

```
interface Calculator
{
   void mul();
   void div();
}
```

How we will write

⇒

```
interface Calculator
{
  public abstract void mul();
  public abstract void div();
}
```

How actually compiler considers inside

## Rule 13:

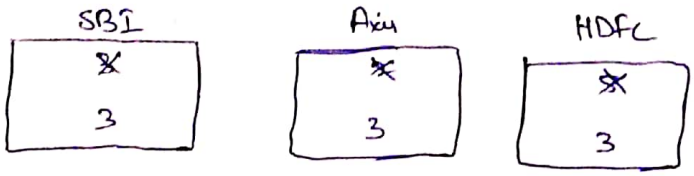
Inside an interface along with methods we can also have variable declaration all variables inside a interface are automatically public, static and final. It is public so that it is accessible to everyone, it is static so that one copy will be shared among multiple objects, it is final so that none of implementing classes can modify values.

```
interface Calculator
{
    void mul();
    void div();
    int count=3;
}
```

⇒

```
interface Calculator
{
    public abstract void mul();
    public abstract void div();
    public static final int count=3;
}
```

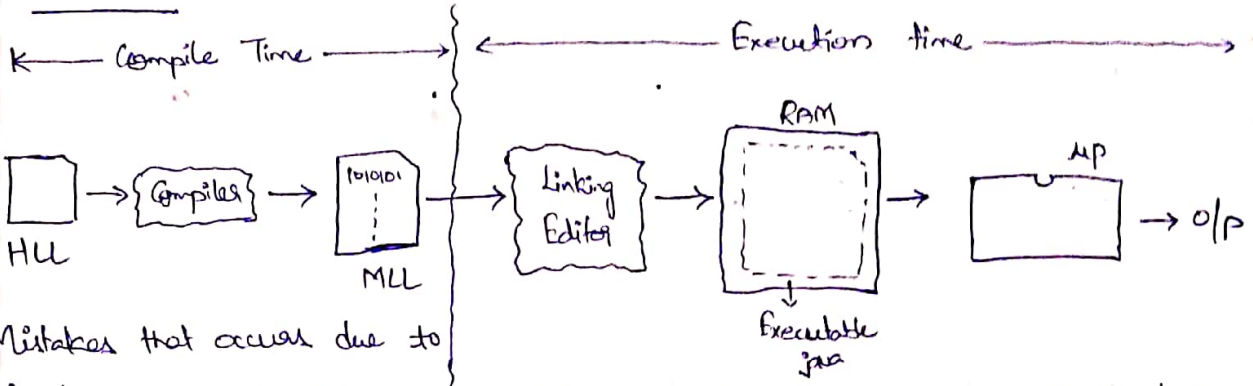
```
int count=3;
```




## Rule 14:

An empty interface is permitted in java. Empty interface is also called as "marker" interface (or) "Tagged" interface.

# EXCEPTIONS:-



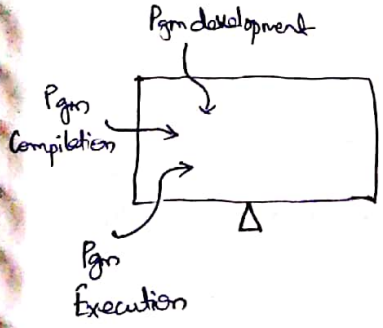

Mistakes that occurs due to faulty usage of language
↳ Compiletime errors

Mistakes that occurs due to faulty logic → Run time error
Mistakes that occurs due to faulty i/p → Exception

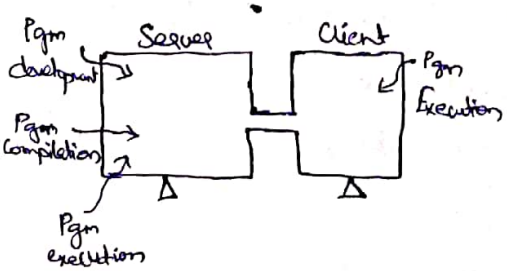
## NOTE:

1) Exception refers to the mistakes that occur during run time due to faulty inputs.
2) Compilation / Compile time error refers to mistakes that occur during compilation time due to faulty usage of language.
3) Run time error refers to mistakes that occur during run time due to improper logic.

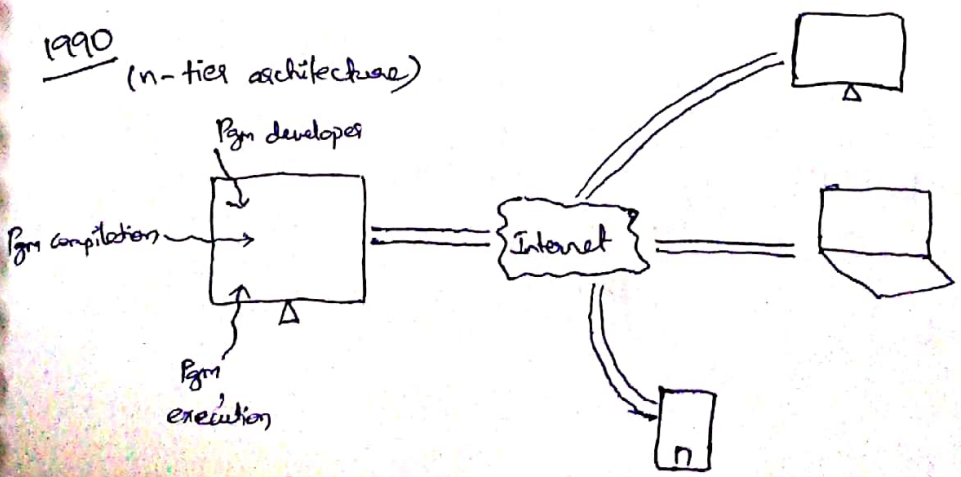
1970 (1-tier architecture standalone architecture)



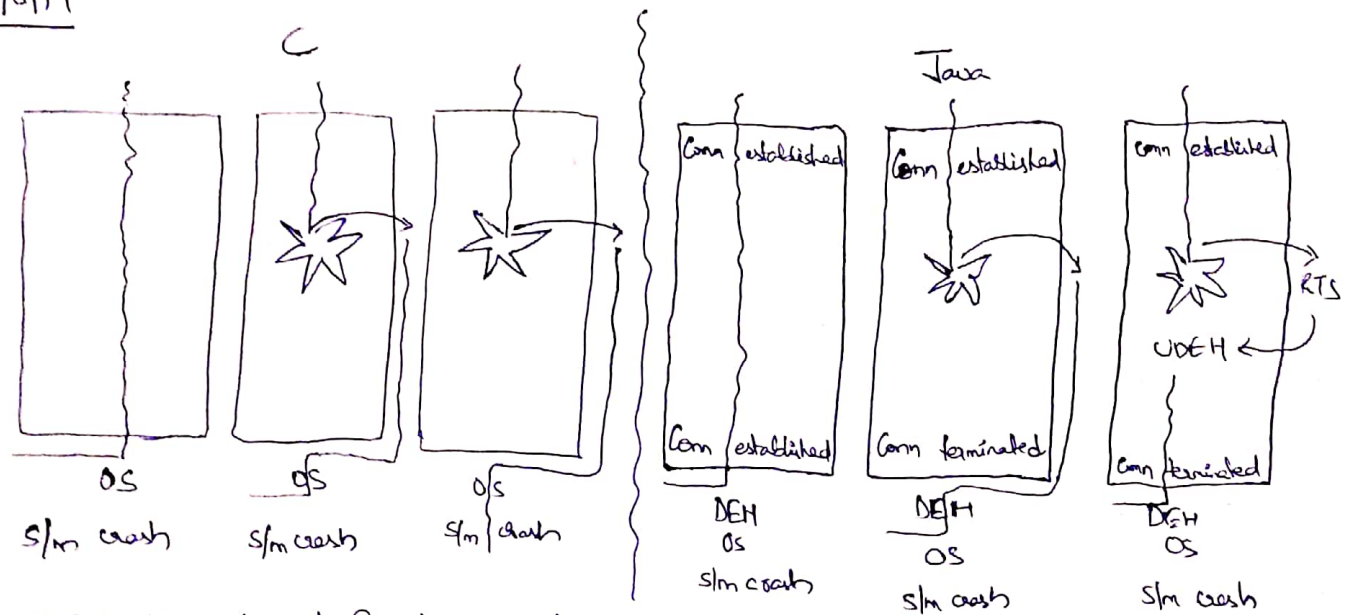

1980 (2-tier architecture server client architecture)




1990 (n-tier architecture)



2/8/19

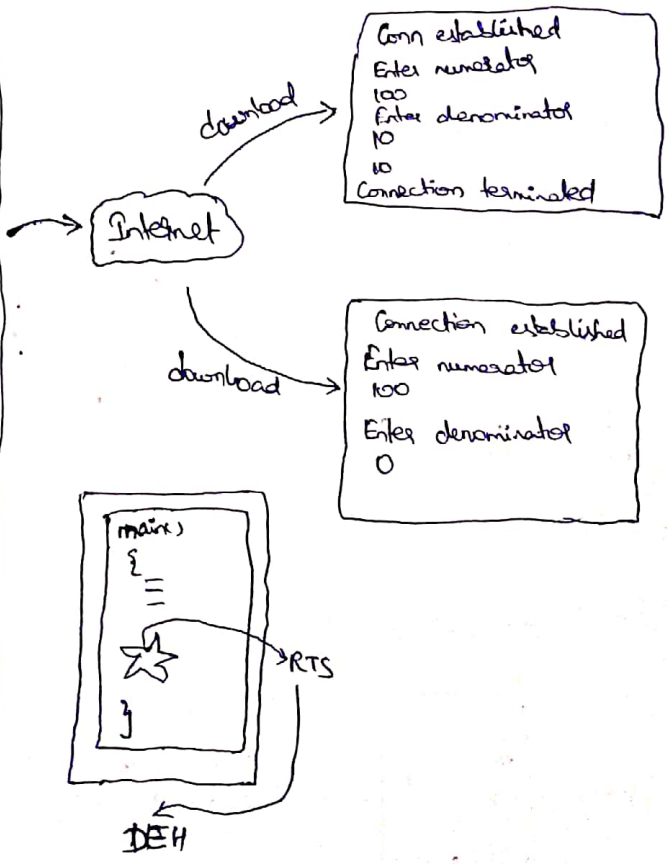



UDEH- User defined Exception Handler
RTS – Run time System
DEH – Default Exception Handler

s/m- system

Exception handling is very much important when it comes to java progeamming, if exceptions couldn't handle it would put so many users under trouble.

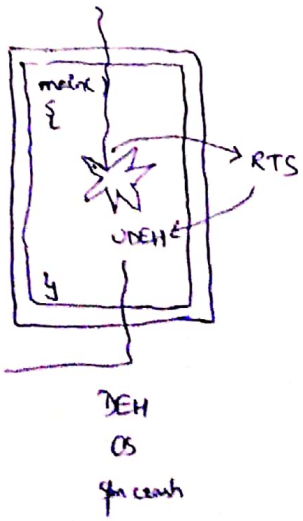
```
class Demo
{
  public static void main(String args[])
  {
    System.out.println("Connection established");
    Scanner scan=new Scanner(System.in);
    System.out.println("Enter a numerator");
    int a=scan.nextInt();
    System.out.println("Enter denominator");
    int b=scan.nextInt();
    int c=a/b;
    System.out.println(c);
    System.out.println("Connection terminated");
  }
}
```

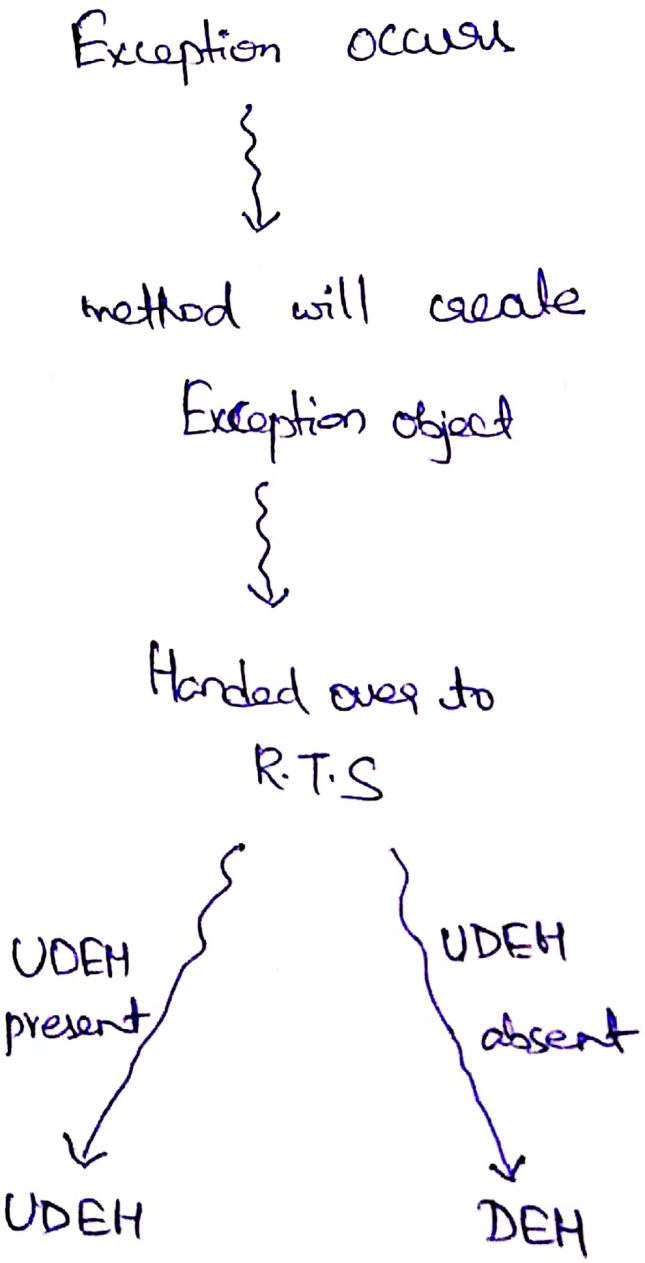


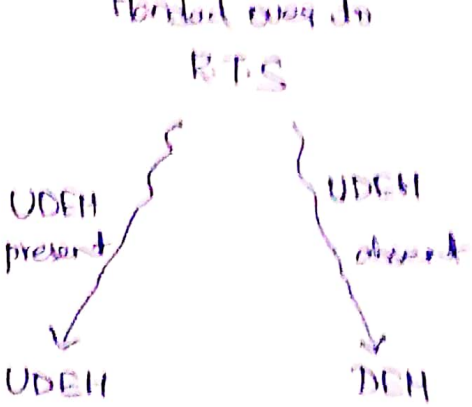
Exceptions will always result in abrupt termination of application.

In java DEH is already been provided by James Gosling, However DEH will ensure system safety and it will not terminate the connection. If connection has to be terminated, then programmer has to provide user defined exception handler (UDEH).

```
Class Demo
{
  public static void main(String args[])
  {
    System.out.println("Connection established");
    try
    {
     Scanner scan=new Scanner(System.in);
     System.out.println("Enter the numerator");
     int a=scan.nextInt();

     System.out.println("Enter denominator");
     int b=scan.nextInt();
     int c=a/b;

     System.out.println(c);
    }

    Catch(Exception e)
    {
     System.out.println("enter valid inputs");
    }

    System.out.println("connection terminated");
  }
}
```








5/8/19 Program 1 :-

```
import java.util.*;
class Demo
{
  public static void main(String args[])
  {
    System.out.println("Connection established");
    try
    {
      Scanner scan = new Scanner(System.in);
      System.out.println("Enter the numerator");
      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 size of array");
      int size = scan.nextInt();
```

```
int aa[]=new int[size];
System.out.println("Enter position of element to be inserted")
int pos= scan.nextInt();
System.out.println("Enter the element to be inserted");
int elem = scan.nextInt();
aa[pos]=elem;
}
catch(Exception e)
{
System.out.println("Invalid Inputs");
}
System.out.println("Connection terminated");
}
}
```

Program 2 :-

```
import java.util.Scanner;
class Demo
{
public static void main(String args[])
{
System.out.println("Connection established");
try
{
Scanner scan=new Scanner(System.in);
System.out.println("Enter the numerator");
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 array");
int size=scan.nextInt();
int aa[]=new int[size];
```

```
System.out.println("Enter the position of element to be inserted");
int pos=scan.nextInt();
System.out.println("Enter the element to be inseated");
int element=scan.nextInt();
  a[pos]=elem;
catch (Arithmetic Exception e)
{
  System.out.println ("Enter the valid denominator");
}
catch(NegativeArraySizeException f)
{
  System.out.println(" Be positive");
}
catch(ArrayIndexOutOfBoundsException g)
{
  System.out.println(" Be in your limit");
}
catch(Exception h)
{
  System.out.println ("Invalid Inputs");
}
System.out.println("Connection Terminated");
}}
```

NOTE:-

1) whenever exception is generated, a method in which exception has occured will create an exception object and hands it over to our time, RTS will search user define exception handler (UDEH) and if it is found it will handover exception object to UDEH and if it is not found it would handover exception to default exception handler.

Advantage of default exception handler (DEH):

It will display correct message to the mistake committed by user.
However disadvantage is it will not terminate the connection.

Advantage / Disadvantage of UDEH:

The advantage of UDEH it will terminate the connection.
However disadvantage is for every type of mistake it will display same message.

→ The problem associated single catch block (i.e program 1) can be overcome by using multiple catch blocks

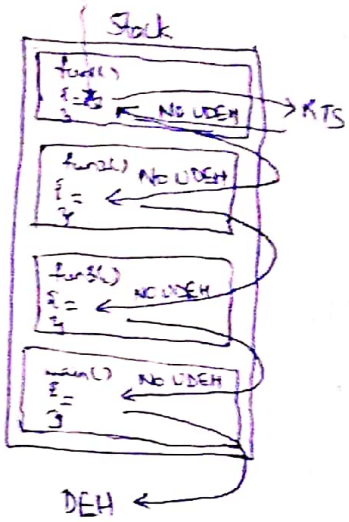
NOTE:

(1) As a developer we should anticipate all types of mistakes that a user may commit and accordingly specific catch block should be provided.
(2) As a precautionary measure a generic catch block should be included at the bottom of specific catch block. If none of specific catch block can't handle the situation then generic catch block will handle.
(3) Exception type refers to all type of exception because exception is parent for all exceptions.
(4) A generic catch block should never be included at the top of other catch blocks because generic catch block has the ability to catch all types of exception.

6/8/19

Propagation of exception:

```
import java.util.*;
class Demo1
{
  void fun1()
  {
   Scanner scan=new Scanner(System.in);
   System.out.println("Enter the numerator");
    int a= scan.nextInt();
   System.out.println("Enter the denominator");
    int b=scan.nextInt();
    int c= a/b;
   System.out.println(c);
  }
}
class Demo2
{
  void fun2()
  {
   Demo1 d1=new Demo1();
    d1.fun1();
  }
}
class Demo3
{
  void fun3()
  {
   Demo2 d2=new Demo2();
      d2.fun2();
  }
}
class LaunchDemo
{ public static void main(String args[])
  {
    Demo3 d3=new Demo3();
  } d3.fun3();
}
```




O/p: ArithmeticException / by zero

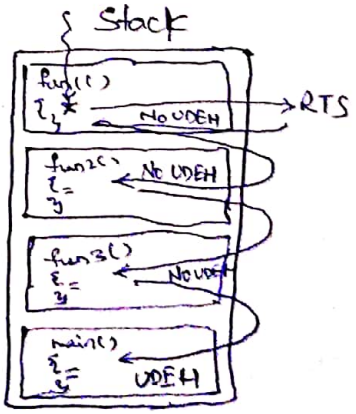
at Demo.fun1();
Demo.fun2();
Demo.fun3();
Demo.main();

NOTE:

1) If exception occurs in one of activation records and if there is no user defined exception handler in that activation record then R.T.S will not immediately hand it over to default exception handler rather it will propagate the exception object to check if there is any handler provided in remaining activation records. If none of the activation record contains UDEH only then the exception object will be given to DEH.

```
class Demo1
{
  void fun1()
  {
    =
  }
}
class Demo2
{
  void fun2()
  {
    =
  }
}
class Demo3
{
  void fun3()
  {
    =
  }
}
```

```
class LaunchDemo
{
  public static void main(String args[])
  {
    try
    {
      Demo3 d3=new Demo3();
      d3.fun3();
    }
    catch(ArithmeticException e)
    {
      System.out.println("Problem solved in main");
    }
  }
}
```



Different ways to managing an exception:

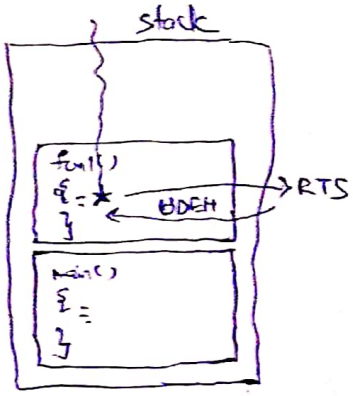
(1) Handling an exception (by using try and catch)

(2) Rethrowing an exception (by using try, catch, throw, throws and finally)

(3) Ducking an exception (by using try, catch and throws)

(1) Handling an exception:

```
import java.util.*;
class Demo1
{
  void fun1()
  {
   System.out.println("Connection2 established");

   Scanner scan=new Scanner(System.in);
   try
   {
    System.out.println("Enter numerator");
    int a=scan.nextInt();

    System.out.println("Enter denominator");
    int b=scan.nextInt();
    int c=a/b;

    System.out.println(c);
   }
   catch(ArithmeticException e)
   {
     System.out.println("Problem resolved in fun()");
   }
   System.out.println("Connection2 terminated");
  }
}
```



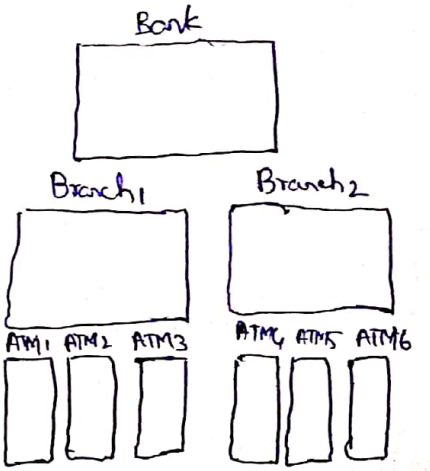
```
class LaunchDemo
{
  public static void main(String args[])
  {
    System.out.println("Connection1 established");
      Demo d1=new Demo1();
      d1.fun();
    } System.out.println("Connection1 terminated");
  }
```

NOTE:-

(1) In the above program exception occured in fun1() method and exception was handled in fun1() only. The main method is not aware of the exception that occured in fun1() this is because whenever exception is handled the information about an exception will not be automatically propagated to other methods.

(2) Rethrowing an Exception:-

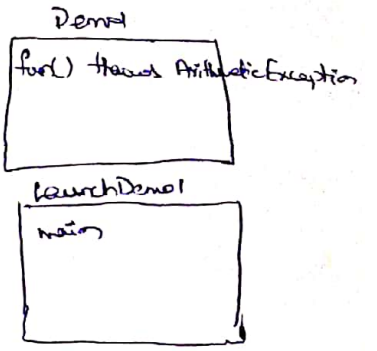
```
class Demo1
{
  void fun1()
  {
   System.out.println("Connection2 established");
   Scanner scan=new Scanner(System.in);
   try
   {
    System.out.println("Enter numerator");
     int a=scan.nextInt();
    System.out.println("Enter deronimator");
     int b= scan.nextInt();
     int c=a/b;
     System.out.println(c);
   }
   catch(ArithmeticException e)
   {
     System.out.println("Problem resolved in fun1");
     throw e;
   }
```



```
finally
{
  System.out.println("Connection2 terminated");
}
}
class LaunchDemo
{
  public static void main(String args[])
  {
    System.out.println("Connection1 established");
    try
    {
      Demo1 d1=new Demo1();
      d1.fun();
    }
    catch(ArithmeticException e)
    {
      System.out.println("Problem resolve in main");
    }
    System.out.println("Connection1 terminated");
  }
}
```

7/8/19

```
import java.util.Scanner;
class Demo1
{
  void fun1()  throws ArithmeticException
  {
    System.out.println("Connection2 established");
    Scanner scan=new Scanner(System.in);
    try
    {
      System.out.println("Enter numerator");
        int a=scan.nextInt();
```




```
System.out.println("Enter denominator");
  int b=scan.nextInt();
  int c=a/b;
  System.out.println(c);
}
catch(ArithmeticException a)
{
  System.out.println("problem resolved in fun");
  throw a;
}
finally
 {
  System.out.println("Connection terminated");
 }
}
}
class LaunchDemo1
 {
   public static void main(String args[])
   {
     System.out.println("Connection1 established");
     try
     {
       Demo1 d1=new Demo1();
       d1.fun1();
      }
     catch(ArithmeticException e)
     {
       System.out.println("Problem resolved in main");
      }
     finally
     {
       System.out.println("Connection1 terminated");
      }
}
}
```

NOTE:

(1) If the same method throws multiple exception you can use multiple catch blocks (or) one generic catch block. And also in method you need to write multiple exceptions like

```
void fun() throws ArithmeticException, NegativeArraySizeException
```

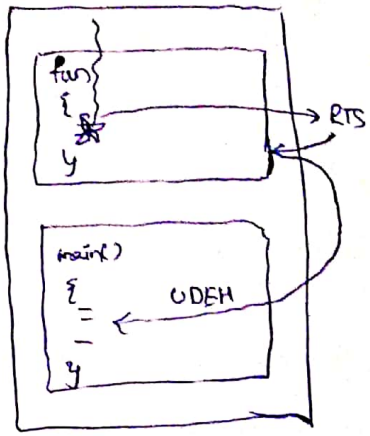
(2) Whenever we want to propagate an exception object we should make use of 'throw' keyword. The disadvantage of throw keyword is the lines below throw keyword will not executed, however we can overcome this problem by using finally block. If there is any critical statement which compulsorily have to be executed such kind of statements have to be in finally block

finally block should get executed even if there is a return statement.

(3) Ducking an exception:

```
import java.util.Scanner;
class Demo1
{
  void fun1() throws ArithmeticException
  {
    Scanner scan=new Scanner(System.in);
    System.out.println("Enter numerator");
    int a= scan.nextInt();
    System.out.println("Enter denominator");
    int b= scan.nextInt();
```

```
        int c=a/b;
        System.out.println(c);
    }
}
class LaunchDemo
{
  public static void main(String args[])
  {
    System.out.println("Connection established");
    try
    {
      Demo1 d1=new Demo1();
      d1.fun1();
    }
    catch(ArithmeticException e)
    {
      System.out.println("Problem resolved in main");
    }
    System.out.println("Connection terminated");
  }
}
```




Note:

(1) Rethrowing an exception refers to the process of handling an exception and then propagating exception object to the other methods.

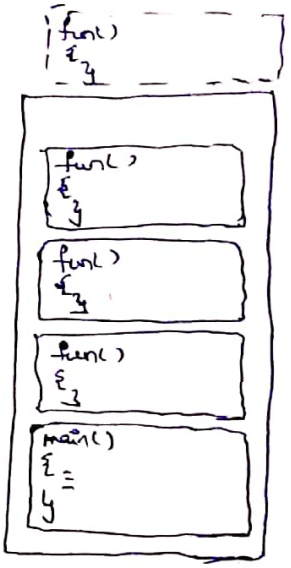
2) Ducking an exception refers to passing the exception to other methods without handling it.

Difference between throw and throws

| throw | throws |
|---|---|
| (1) throw keyword is used for rethrowing an exception. | (1) throws keyword is used for ducking exception |
| (2) throw keyword is used within body of method. | (2) throws keyword will be used in signature of the method. |
| (3) The lines below throw keyword will not get executed. | (3) The lines below throws keyword will get executed. |

RUN TIME ERROR:-

(1)
```
class Demo1
void fun()
{
  fun();
}
class LaunchDemo1
{
 public static void main(String args[])
 {
   Demo1 d1=new Demo1();
   d1.fun()
 }
}
```




O/p:- Error: StackOverFlow

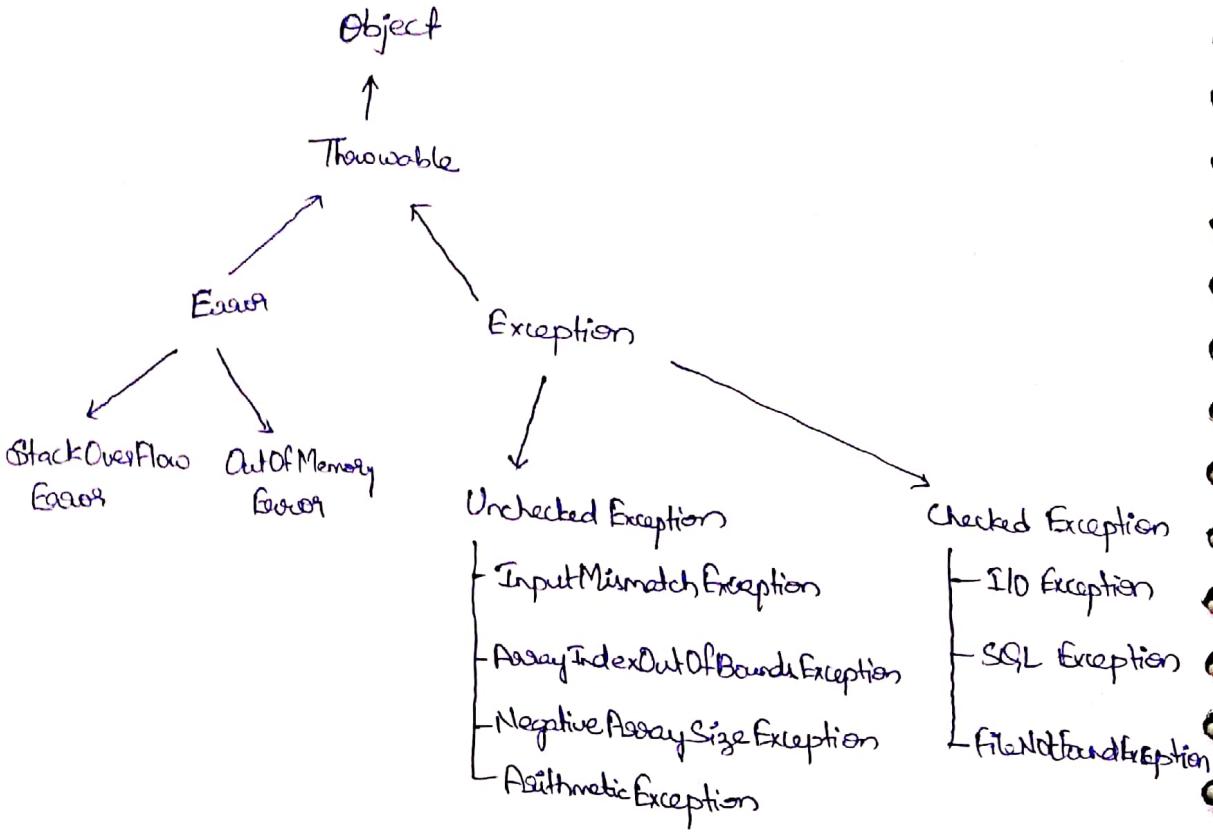
(2)
```
class Demo1
void fun()
{
  try
  {
   fun();
  }
  catch (Exception e)
  {
   System.out.println("Problem resolved in fun");
```

```
    }
}
class LaunchDemo1
{
  public static void main(String args[])
  {
    Demo1 d1=new Demo1();
    d1.fun();
  }
}
```

8/8/19

Difference Runtime error and Exception

| Runtime error | Exception |
|---|---|
| (1) They are the mistakes that occur during runtime due to improper logic by programmers. | (1) These are the mistakes that occur during execution due to faulty inputs provided. |
| (2) They cannot be managed using try and catch. | (2) They can be managed using try and catch. |
| (3) Types/Examples of errors are StackOverflow Error and OutOfMemoryError | (3) Types of exceptions are checked exceptions and unchecked exceptions. |



9/8/19

Unchecked Exceptions: Unchecked are such exceptions where the compiler doesnot force the programmer to handle the exceptions.

Checked Exceptions: Checked exceptions are such exceptions where compiler forces the programmer to handle the exceptions

Lisko Substitution Rules:

Rule 1: If a method in parent class throws an exception then it is not compulsory for the overridden in child class to throw an exception.

Ex:
```
class Demo
{
  void disp() throws ArithmeticException
  {
    =
  }
}
class Demo2 extends Demo1
{
  void disp() throws ArithmeticException   (or)   void disp()
  {                                                 {
    =                                                 =
  }                                                 }
```

Rule 2: If a method in parent class throws one type of exceptions and a overridden method in child class throws another type of exception then it is not permitted.

```
class Demo
{
  void disp() throws ArithmeticException
   {
    =
   }
}
class Demo2 extends Demo1
{
  void disp() throws SQLException
   {
    =
   }
}
```

Rule 3: If a method in parent class throws one type of exceptions and a overridden method in child class throws another type of exception it is still permitted if there is is-a relationships blw them.

```
class Demo0
{
  void disp() throws Exception
   {
    =
   }
}
class Demo2 extends Demo1
{
  void disp() throws SQLException
   {
    =
   }
}
```

is a relation so acceptable

If a method in parent class throws one type of exception and a overridden method in child class throws another (type) of exception even if is-a relationship doesnot exist still it is permitted if the exception are of type unchecked exceptions.

```
class Demo1
{
  void disp() throws InputMisMatchException
  {
  =
  }
}
class Demo2 extends Demo1
{
  void disp() throws ArrayIndexOutOfBoundsException
  {
  =
  }
}
```

---

Customized Exceptions:

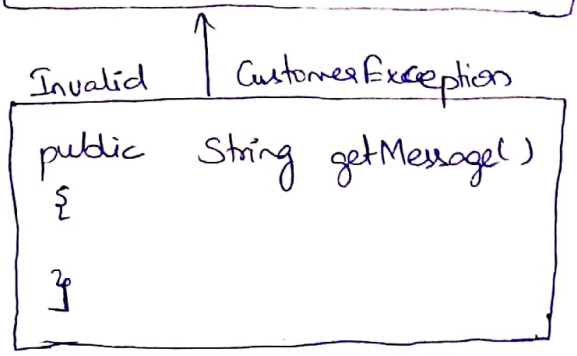
```
class Atm
{
  int accNO=111;
  int pwd=2222;
  int acno;
  int pass;

Scanner scan=new Scanner(System.in);
void input()
{
  System.out.println("Enter the accno");
   acno=scan.nextInt();
  System.out.println("Enter password");
   pass=scan.nextInt();
}
```

```
void verify()
 {
   if( accno==acno && pwd==pass)
    System.out.println("Collect your cash");
   else
    System.out.println("Enter valid inputs");
  }
}

class Bank
 {
   void initiate()
   {
     Atm a=new Atm();
     a.input();
     a.verify();
   }
 }
 class LaunchBank
 {
  public static void main(String args[])
  {
    Bank b=new Bank();
    b.initiate();
  }
 }
```

14/8/19

Exception

```
public String getMessage()
{
  return null;
}
public void printStackTrace()
{

}
```




```
public String getMessage()
{

}
```

```
class InvalidCustomerException extends Exception
{
  public String getMessage()
  {
    return "Invalid customer pls enter valid inputs";
  }
}
class Atm
{
  int accno=111;
  int pwd=2222;
  int acno;
  int pwd;
  void input()
  {
    Scanner scan=new Scanner(System.in);
    System.out.println("Enter accno");
    acno=scan.nextInt();
    System.out.println("Enter pwd");
    pass=scan.nextInt();
  }
```

```
void verify() throws InvalidCustomerException
{
  if(accno==acno && pwd==pass)
  {
   System.out.println("Collect cash");
  }
  else
  {
   InvalidCustomerException ice=new InvalidCustomerException();
   System.out.println(ice.getMessage());
   throw ice;
  }
}
class Bank
{
  void intiate()
  {
    Atm a=new Atm();
    try
    {
      a.input();
      a.verify();
    }
    catch(InvalidCustomerException ice)
    {
     System.out.println(ice.getMessage());
    }
  }
}
```

```
class LaunchBank
{
  public static void main(String args[])
  {
    Bank b=new Bank();
    b.intiate();
  }
}
```

Valid Syntaxes:

```
→ try
  {

  }
  catch( )
  {

  }
```

```
→ try { }
  catch( ) { }
```

```
→ try {
  }
  catch( ) {

  }
```

```
→ try
  {

  }
  catch( )
  {

  }
  finally
  {

  }
```

```
→ try
  {
    try
    {

    }
    catch( )
    {

    }
    finally
    {

    }
  }
  catch( )
  {

  }
  finally
  {

  }
```

```
→ try
  {
  }
  catch( )
  {
  }
  catch( )
  {
  }
  finally
  {
  }
```

→
```
try
{
}
catch()
{
  try
  {
  }
  catch()
  {
  }
  finally
  {
  }
}
finally
{
}
```

→
```
try
{
}
catch()
{
}
finally
{
  try
  {
  }
  catch()
  {
  }
  finally
  {
  }
}
```

## Invalid Syntaxes:

→
```
try
{
}
```

→
```
catch
{
}
```

→
```
finally
{
}
```

→
```
try
{
}
finally
{
}
```

NOTE:

Custom exception refers to user defined exceptions. In order to create custom exceptions we have to follow two steps.

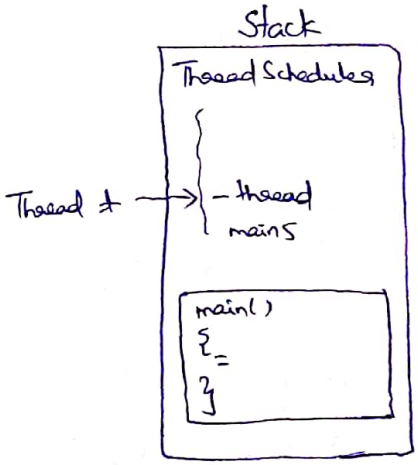
→ Our custom exception should be a child class of exception class.

→ We have to override getMessage method.

# MULTI-THREADING

## Thread:

```
class Demo
{
  public static void main(String args[])
  {
    System.out.println("Good Morning");
    Thread t=Thread.currentThread();
    System.out.println(t);
    t.setName("ABC");
    t.setPriority(7);
    System.out.println(t);
  }
}
```




O/p:- Good morning

Thread[main,5,main]

Thread[ABC, 7, main]

## Note:

→ Thread is nothing but sequence of execution.

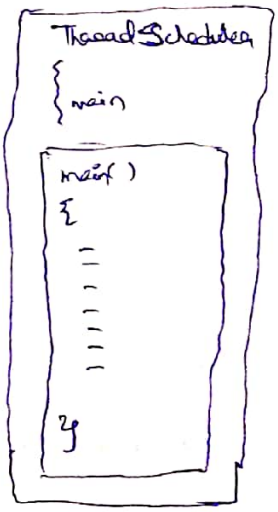
→ Automatically for every program there's a thread created and it is called as main thread. The priority given for main thread is 5.

→ We can change the name of the thread by using setName() and we can change the priority of a method by using setPriority().

15/8/19

## Disadvantage of Single thread programming:

```
import java.util.*;
class Demo
{
  public static void main(String args[]) throws Exception
  {
   System.out.println("Banking activity started");
   Scanner scan=new Scanner(System.in);
   System.out.println("Enter the acno");
    int acno=scan.nextInt();
  System.out.println("Enter the password");
   int pwd=scan.nextInt();
  System.out.println("Collect cash");
  System.out.println("Banking activity completed");
  System.out.println("Painting activity started");
  for(int i=1;i<=5;i++)
  {
    System.out.println("ABC");
    Thread.sleep(5000);
  }
  System.out.println("Painting activity completed");
  System.out.println("addition activity started");
   int a=123;
   int b=456;
    int c=a+b;
     System.out.println(c);
    System.out.println("addition activity completed");
  }
}
```



```
import java.util.*;
class Demo1
{
  void banking()
  {
    System.out.println("Banking started");
    ==
    System.out.println("Banking completed");
  }
  void printing()
  {
    System.out.println("Printing started");
    ===
    System.out.println("Painting completed");
  }
  void addition()
  {
    System.out.println("addition started");
    ==
    System.out.println("addition completed");
  }
}
class LaunchDemo
{ public static void main(String args[])
  {
    Demo1 d1=new Demo1();
    d1.banking();
    d1.printing();
    d1.addition();
  }
}
```

NOTE:-

(1) Writing different activities in different methods is not going to give solutions for the problem associated with single threaded programming.

(2) Problem associated with above programming can be overcome by multithreading. There are two ways to create multiple threads
(i) By extending thread class
(ii) By implementing "Runnable" interface.

(i) By extending Thread class:-

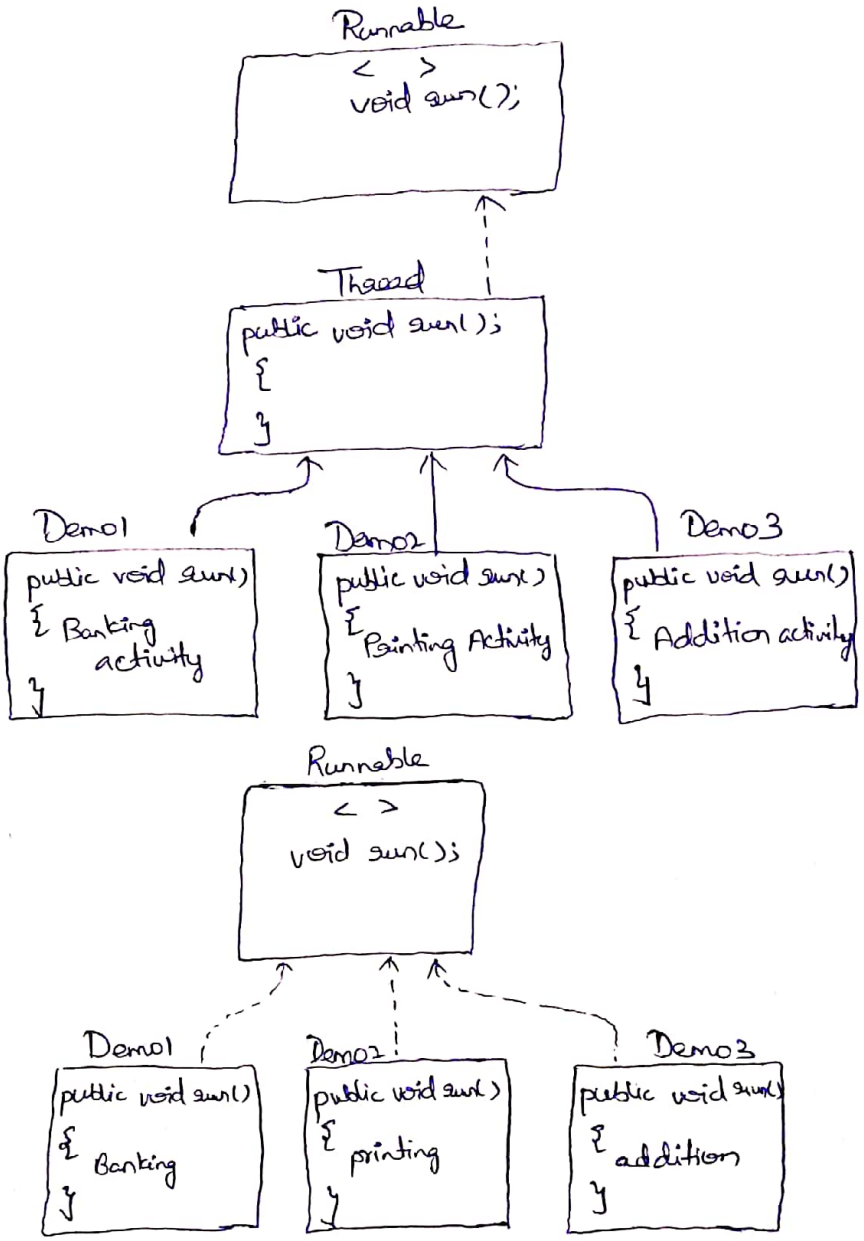
Thread
public void run()
{
}

```
import java.util.*;
class Demo1 extends Thread
{
  public void run()
  {
   System.out.println("Banking started");
   Scanner scan=new Scanner(System.in);
   System.out.println("Enter the acno");
   int acno=scan.nextInt();
   System.out.println("Enter the password");
   int pwd=scan.nextInt();
   System.out.println("Collect cash");
   System.out.println("Banking activity completed");
  }
}
```

```
class Demo2 extends Thread
{
  public void run()
  {
    System.out.println("Painting activity started");
    for(int i=1; i<=5; i++)
    {
      try
      {
        System.out.println("ABC");
        Thread.sleep(5000);
      }
      catch (Exception e)
      {
        System.out.println("Problem resolved");
      }
    }
    System.out.println("Printing activity completed");
  }
}

class Demo3 extends Thread
{
  public void run()
  {
    System.out.println("Addition activity started");
    int a=123;
    int b=989;
    int c=a+b;
    System.out.println(c);
    System.out.println("Addition activity completed");
  }
}
```

```
class LaunchDemo
{
   public static void main(String args[])
   {
     Demo1 d1=new Demo1();
     Demo2 d2=new Demo2();
     Demo3 d3=new Demo3();
     d1.setName("Banking");
     d2.setName("Painting");
     d3.setName("Addition");
     d1.start();
     d2.start();
     d3.start();
   }
}
```

16/8/19




```
class Demo1 implements Runnable
{
    public void run()
    {
      System.out.println("Banking started");
```

}

}

```
class Demo2 implements Runnable
{
  public void run()
  {
    System.out.println("Printing started");

  }
}

class Demo3 implements Runnable
{
  public void run()
  {
    System.out.println("addition started");

  }
}
```

```
class LaunchDemo
{
 public static void main(String args[])
 {
  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());
  System.out.println(t2.isAlive());
  System.out.println(t3.isAlive());
  t1.start();
  t2.start();
  t3.start();
```

Note:

In the above program creating an object of demo1, demo2, demo3 will not create extra threads because there is no relation between demo1, demo2, demo3 and thread class.

Therefore in order to create additional threads we have to create object for thread explicitly.

After creating the thread we have to assign suitable work to thread.

Note:

(1) It is always main thread who will finish execution first. If main thread has to wait for other threads to complete the work then we have to make use of join method

(2) The thread will not be alive so after creation, it will come to live only after starting the method.

19/8/19

```
class MsWord extends Thread
{
  public void run()
  {
    if(getName().equals("TYPING"))
      typing();
    else if(getName().equals("AUTOSAVING"))
      autoSaving();
    else
      spellChecking();
  }
  void typing
  {
    for(int i=1;i<=5;i++)
    {
      try
      {
        System.out.println("Typing");
        Thread.sleep(5000);
      }
    }
      catch(Exception e)
      {
        System.out.println("Typing Interrupted");
      }
  }  ~~System.out.pri~~
  void autoSaving
  {
    for(int i=1; i<=5; i++)
    {
      try
      {
        System.out.println("autosaving");
        Thread.sleep(5000);
```

```
        }
      }
      catch (Exception e)
      {
        System.out.println("autosaving interrupted");
      }
    }
    void spellChecking()
    {
      for (int i=1; i<=5; i++)
      {
        try
        {
          System.out.println("Spellchecking");
          Thread.sleep(5000);
        }
      }
      catch(Exception e)
      {
        System.out.println("spell checking interrupted");
      }
    }
class LaunchMsWord
{
  public static void main(String args[])
  {
    MsWord ms1=new MsWord();
    MsWord ms2=new MsWord();
    MsWord ms3=new MsWord();
    ms1.setName("TYPING");
    ms2.setName("AUTOSAVING");
    ms3.setName("SPELLCHECKING");
    ms1.start();
    ms2.start();
    ms3.start();
  }
}
```

NOTE:-

(1) In the above program typing activity is main activity where as autosavi and spell checking are secondary activities.

(2) Auto saving and spell checking should wait for typing activity to get over however in above program all the three threads are in hurry to complete their work. The secondary activities should get executed atleast once after the primary activity is completed and such threads are called "Daemon Threads".

(2)
```
class MsWord extends Thread
{
    public void run()
    {
      if(getName().equals("TYPING"))
         typing();

      else if(getName().equals("AUTOSAVING"))
         autoSaving();
      else
         spellChecking();

    }
    void typing
    {
       for(int i=1; i<=5; i++)
       {
          try
          {
             System.out.println("Typing");
          } Thread.sleep(5000);
          }
          catch(Exception e)
          {
            System.out.println("Typing Interrupted");
    } }
```

```
void autoSaving
{
  for( ; ; )
  {
    try
    {

    }
    catch( )
    {

    }
  }
}

void spellChecking()
{
  for( ; ; )
  {
    try
    {

    }
    catch( )
    {

    }
  }
}
}

class LaunchMsWord
{ public static void main(String args[ ])
  {
     MsWord ms1=new MsWord( );
     MsWord ms2=new MsWord( );
     MsWord ms3=new MsWord( );

     ms1.setName("TYPING");
     ms2.setName("AUTOSAVING");
     ms3.setName("SPELLCHECKING");
     ms2.setPriority(7);
     ms3.setPriority(8);
```

```
        ms2.Daemon(true);
        ms3.Daemon(true);
        ms1.start();
        ms2.start();
        ms3.start();
    }
}
```

Note:

(1) For making threads as daemon threads, you should follow these steps

(i) Secondary activities should be enclosed with in infinite loops.

(ii) Secondary threads should be given low priority (anything above 5)

(iii) We have to set the status of setDaemon as true.

Ex: setDaemon(true);

class Demo extends Demo2, Thread

↓

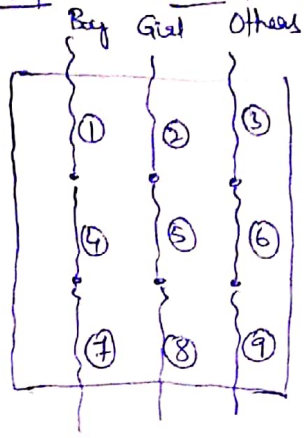
XX → It will give error because it is multiple inheritance.

class Demo extends Demo2 implementing Thread

↓

√√ → It is the best approach to achieve multithreading by implementing interface.

20/8/19 Problem associated with multithreading:




```
class Bathroom implements Runnable
{
  public void run()
  {
    try
    {
      System.out.println(Thread.currentThread().getName()+" entered Bathroom");
      Thread.sleep(5000);
      System.out.println(Thread.currentThread().getName()+" started using Bathroom");
      Thread.sleep(5000);
      System.out.println(Thread.currentThread().getName()+" exits from Bathroom");
      Thread.sleep(5000);
    }
    catch(Exception e)
    {
      System.out.println("Interrupted");
    }
  }
}
```

```
class LaunchBathroom
{
  public static void main(String 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("OTHERS");
    t1.start();
    t2.start();
    t3.start();
  }
}
```

NOTE:

1) In the above program a single ~~thread~~ resource has been shared among multiple threads. If single resource has to be shared among multiple threads, then multi threading is not a good approach.
If one thread gets a chance and fails for some reason, then thread schedules in order to utilize CPU time efficiently, it will schedule another thread and hence it is not good approach.

2) Above problem can be overcome by using two ways
   i) By using "Synchronized keyword".

```
class Bathroom implements Runnable
{
   synchronized ~~void~~ public void run()
   {
     try
     {
        -
        -
        -
        -
     }
     catch(Exception e)
     {
        -
        -
     }
   }
}

class LaunchBathroom
{
    -
    -
    -
    -
}
```

In the above program whichever thread gets the chance will start executing the run() method and if the existing ~~method~~ thread halts due to some reason, even then the thread scheduler (TS) will not be able to another thread. Only after executing thread completely finishes the work, the Thread Scheduler will be able to schedule another thread. This is technically called as "Semaphore or monitor".

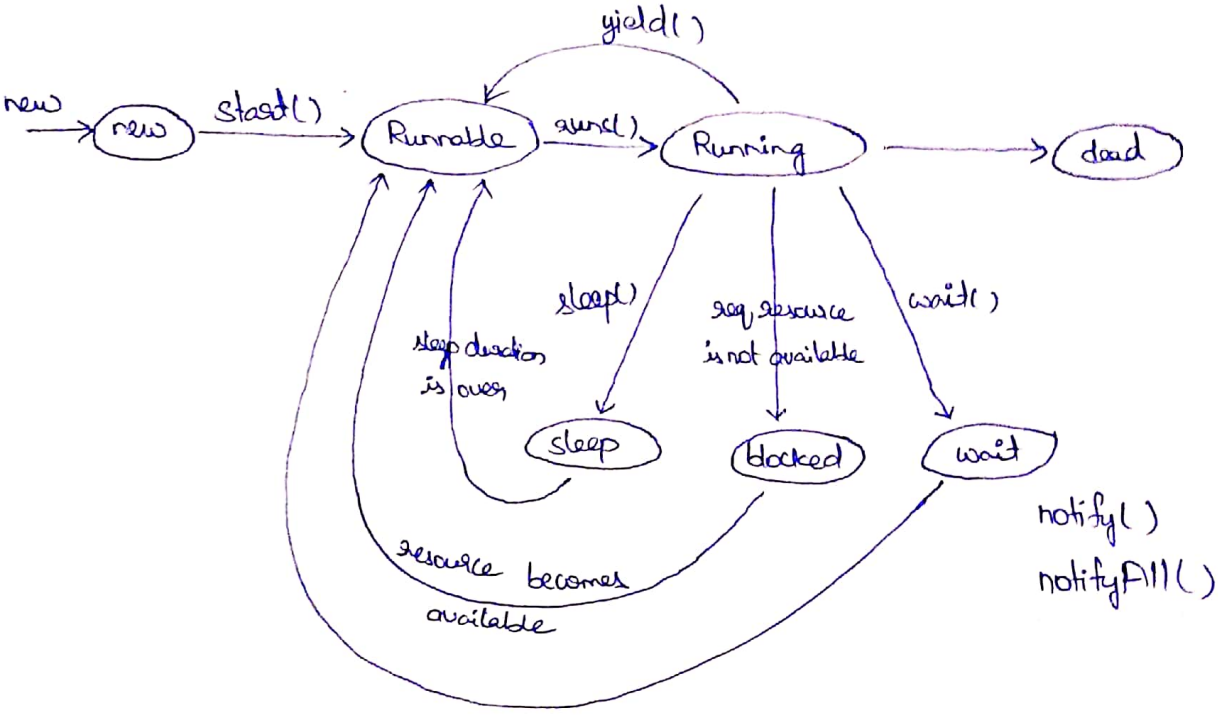
ii) by using join() method:-

```
class Bathroom implements Runnable
{
   public void run()
   {
      =
      =
      =
   }
}
class LaunchBathroom
{
   public static void main(String 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.join();
      t2.start();
      t2.join();
      t3.start();
      t3.join();
   }
}
```

21/8/19

Even though it (join()) can overcome the problem associated with multi threading still it is not a good approach because the order of thread scheduling will be completely controlled by programmer himself hence it is suggested to overcome the problem by synchronized keyword.

## Different states of thread:




## Demostration of dead lock:

Program 1: Program without deadlock

```
class Warrior extends Thread
{
    String res1="Bhahmastra";
    String res2="Pashupatstra";
    String res3="Sarpastra";
```

```
public void run()
{
  if(getName().equals("RAMA");
      ramaAccRes();
  else
     ravanaAccRes();
 }
void  ramaAccRes()
{
  try
  {
   Synchronized(res1)
   {
    System.out.println("Rama acquires "+res1);
     Thread.sleep(5000);
    Synchronized(res2)
     {
       System.out.println("Rama acquires "+res2);
        Thread.sleep(5000);
      Synchronized(res3)
      {
        System.out.println("Rama acquires "+res3);
         Thread.sleep(5000);
      }
    }
   }
  }
  catch(Exception e)
  {
   System.out.println("Interrupted");
  }
}
```

```
void ravanaAction()
{
  try
  {
    Synchronized(res1)
    {
      System.out.println("Ravana acquires "+res1);
      Thread.sleep(5000);
      Synchronized(res2)
      {
        System.out.println("Ravana acquires "+res2);
        Thread.sleep(5000);
        Synchronized(res3)
        {
          System.out.println("Ravana acquires "+res3);
          Thread.sleep(5000);
        }
      }
    }
  }
  catch(Exception e)
  {
    System.out.println("Interrupted");
  }
}
class LaunchWarrior
{
  public static void main(String args[])
  {
    Warrior w1=new Warrior();
    Warrior w2=new Warrior();
    w1.setName("RAMA");
    w2.setName("RAVANA");
    w1.start();
    w2.start();
  }
}
```

22/8/19

Program 2 : Program using deadlock.

```
class Warrior extends Thread
{
  String res1 = "Brahmastra";
  String res2 = "Pasupatastra";
  String res3 = "Sarpastra";

  public void run()
  {
    if(getName().equalsTo("RAMA")
     {
        ramaAccRes1();
     }
    else
        ravanaAccRes1();

  }

  void ramaAccRes1()
  {
     try
     {
       Synchronized(res1)
       {
         System.out.println("Rama Acquires "+res1);
         Thread.sleep(5000);

       Synchronized(res2)
       {
         System.out.println("Rama Acquires "+res2);
         Thread.sleep(5000);
```

```
Synchronized(res3)
{
  System.out.println("Rama acquires "+res3);
  Thread.sleep(5000);
 }
 }
 }
}
catch(Exception e)
 {
  System.out.println("Interrupted");
 }
}

void ravanaAccRes()
{
  try
   {
    Synchronized(res1)
    {
     System.out.println("Ravana acquires "+res1);
     Thread.sleep(5000);

    Synchronized(res2)
    {
     System.out.println("Ravana acquires "+res2);
     Thread.sleep(5000);

    Synchronized(res3)
    {
     System.out.println("Ravana acquires "+res3);
     Thread.sleep(5000);
```

```
        }
      }
    }
  }
  catch(Exception e)
  {
    System.out.println("Interrupted");
  }
}

class LaunchWarrior
{
  public static void main(String args[])
  {
    Warrior w1=new Warrior();
    Warrior w2=new Warrior();
    Warrior w3=new Warrior();
    w1.setName("RAMA");
    w2.setName("RAVANA");
    w1.start();
    w2.start();
  }
}
```

NOTE:-

(1) In program 1, if Rama gets the chance Rama apply lock on all the resource. At that time Ravana thread will be in Blocked state. After Rama thread completes the work and goes to dead state all the locks will be released and hence Ravana thread will come out of Blocked state.

(2) Instead of Rama if Ravana gets the first chance then Ravana will apply lock on all the resources and Rama thread will be blocked state.

NOTE:-

(1) We can have multiple threads in any of the state except Running state. At running state there can be only one thread at any given point of time.

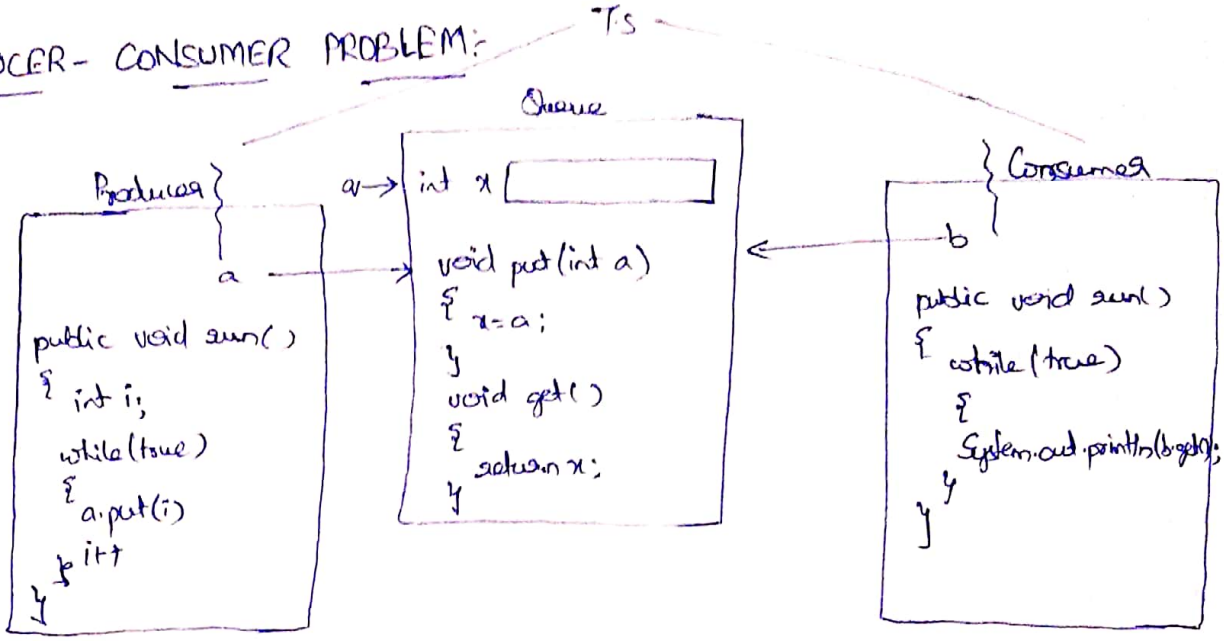
DEADLOCK:-

Deadlock refers to a situation where there are multiple threads in blocked state with mutual dependency.

NOTE:-

(1) In program 2, Ravana will try to acquire the resource in reverse order and hence at some point of time both threads will go to the blocked state. waiting for the resource which the other thread is holding in other words multiple threads present in blocked state is what technically called as deadlock.

# PRODUCER - CONSUMER PROBLEM:




Program 1:- With problem

```
class Producer extends Thread
{
   Queue a;
   int i;

   Producer(Queue q)
   {   a=q;
   }
   public void run()
   {
      int i;
      while(true)
      {
        a.put(i);
        i++;
      }
   }
}
class Queue
{
    int x;
    void put(int a)
    {
      x=a;
    System.out.println(" I have put the value for x "+x);
}
```

```
    void get()
    {
      System.out.println("I have got the value x "+x);
    }
}

class Consumer extends Thread
{
  Queue b;

  Consumer(Queue q)
  {
   b=q;
  }

  public void run()
  {
    while(true)
    {
      b.get();
    }
  }
}

class LaunchProCon
 {
   public static void main(String args[])
   {
     Queue q=new Queue();
     Producer p=new Producer(q);
     Consumer c=new Consumer(q);
      p.start();
      c.start();
}}
```

23/8/19

Program 2:- (Without problem) Solution to producer consumer problem

```
class Queue
{
 int x;
 boolean value_present_in_x=false;
 synchronized void put() throws Exception
 {
  if(value_present_in_x == false)
  {
    x=a;
   System.out.println("I have put the value of x "+x);
   value_present_in_x=true;
    notify();
  }
  else
  {
    wait();
  }
}
synchronized void get() throws Exception
{
  if(value_present_in_x==true)
  {
   System.out.println("I have got the value of x "+x);
   value_present_in_x=false;
    notify();
  }
  else
  {
   wait();
  }
}
```

(Put try & catch in producer class to handle a.put() statement.)

(Put try & catch in consumer class to handle b.get() statement.)

NOTE:-

1) In program 1, there is lack of communication between the producer and consumer thread due to which all the values produced by producer has not been consumed by consumer.

2) Whenever a chance is given to producer it will mechanically keep on producing the values and whenever consumer gets the chance mechanically it keeps on consuming the values. The producer will produce the value without checking consumer has consumed old value (or) not. Similarly a consumer will keep consuming the value without checking whether new value has been produced or not.

NOTE:-

1) In program 2, in order to enable the communication between producer and consumer we have to make use of wait() and notify() methods whenever wait() and notify() methods are used in program, we have to make use of synchronized keyword.

2) Inter-thread communication between threads can be achieved by using wait() and notify() methods.

Intrathread communication between threads can be achieved by synchronized keyword.

# COLLECTIONS HIERARCHY:-

## The disadvantage of Arrays Data Structure:

(i) Arrays cannot store heterogeneous data.

(ii) Arrays sized is fixed once, cannot grow or shrink.

(iii) Arrays store values in continuous memory locations.

→ All the three drawbacks of array data structure can be overcome by using collection classes.

## Different classes present in Collections are:

(1) ArrayList

(2) LinkedList

(3) ArrayDeque

(4) PriorityQueue

(5) TreeSet

(6) HashSet

(7) LinkedHashSet

→ The collection hierarchy was introduced from JDK 1.2. It was a hierarchy given by a freelancer called "Joshua".

## (1) ArrayList:-

ArrayList internally makes use of dynamic array data structure.

```
import java.util.ArrayList
public class Collections
{
    public static void main(String[] args)
    {
        ArrayList al=new ArrayList();
        al.add(10);
        al.add(20.5f);
```

```
        al.add('a');
        al.add("ABC");
        System.out.println(al);
    }
}
```

O/P: [10, 20.5f, a, ABC]

26/8/19

```
import java.util.*;
class Collections
{public static void main(String args[])
 {
   ArrayList al=new ArrayList();
   al.add(10);
   al.add(20);
   al.add(30);
   al.add(40);
   al.add(50);
   System.out.println(al);                    O/P: [10, 20, 30, 40, 50]
   al.add(2,100);
   System.out.println(al);                    O/P: [10,20,100,30,40,50]
   ArrayList al2=new ArrayList();
   al2.add(50);
   al2.add(60);
   al2.add(70);
   System.out.println(al2);                   O/P: [50,60,70]
   al.addAll(al2);
   System.out.println(al);                    O/P: [10,20,100,30,40,50,50,60,70]
```

```
al.addAll(3, al2);
System.out.println(al);          O/P: [10, 20, 100, 50, 60, 70, 30, 40, 50, 50, 60, 70]
al.remove(2);
System.out.println(al);          O/P: [10, 20, 50, 60, 70, 30, 40, 50, 50, 60, 70]
al.retainAll(al2);
System.out.println(al);          O/P: [50, 60, 70, 50, 50, 60, 70]
al.removeAll(al2);
System.out.println(al);          O/P: []
}
}
```

27/8/19

```
import java.util.*;
public class Collections
{
   public static void main(String args[])
   {
    ArrayList al=new ArrayList();
    al.add(10);
    al.add(20);
    al.add(30);
    al.add(40);
    al.add(50);
    al.add(60);
   System.out.println(al);                    O/p:- [10,20,30,40,50,60]
   al.ensureCapacity(8);
   System.out.println(al.size());             O/p:- 6
   System.out.println(al.get(2));             O/p:- 30
   System.out.println(al.getClass());         O/p:- class java.util.ArrayList
   System.out.println(al.indexOf(50));        O/p:- 4
  al.remove(new Integer("50"));               -> O/p:- [10,20,30,40,60]
  System.out.println(al);
```