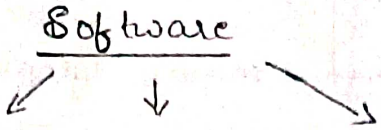
Java Full Stack

5/7/23



Frontend	Backend	Database
HTML5	Java	SQL
CSS3	Adv. Java	
Javascript	J2EE	
React.js	Spring Boot	
Bootstrap	Hibernate	

Software: It is a collection of programs used to provide automated solutions to real world problems.

Program: It is a set of instructions given to a computer in any programming language like C, C++, Java etc.

Journey of computers & programming language:

- Transistors were made up of semiconductors hence they had only 2 states (ON-1, OFF-0).
- Microprocessor which is the brain of the computer is made up of millions of transistors, hence µp understands only Binary language (or) Machine level language (MLL).

⇒ 1940's

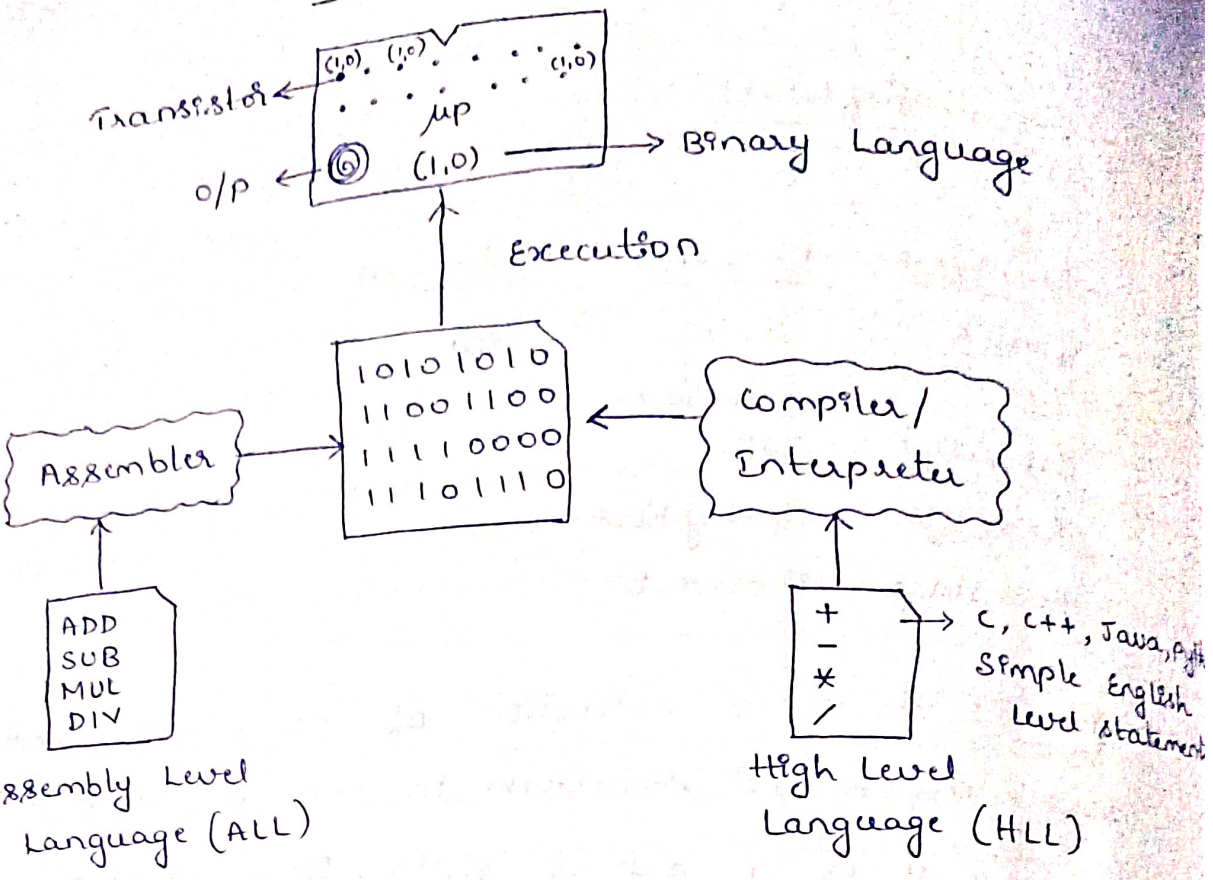
Discovery of Semiconductor.

+10 yrs ↓

A device - Transistor was invented.

Conductor	Insulator
+5V	0V
ON (1)	OFF (0)

Microprocessor (Brain of computer)



Pnemonics
- ADD - (10101010) → Addition
- SUB - (11001100) → Subtraction
- MUL - (11110000) → Multiplication
- DIV - (11101110) → Division

- A language composed of Pnemonics is called as Assembly level language.
- A language that contains special symbols & written using simple english like statements is called High level language.
- Assembler converts ALL to MLL.
- Compiler/Interpreter converts HLL to MLL.

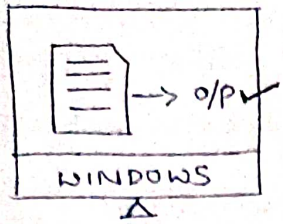
Note:

i) 1st HLL → BCPL (Basic computer Programming language.) → unstructured, Huge memory
ii) B → unstructured, Less memory
iii) (1972) → C → Structured, Less memory
iv) (1982) → C++ → OOPA, Less memory
v) C++++-- → Portable, OOPA, Less memory

(i–iii: not portable)

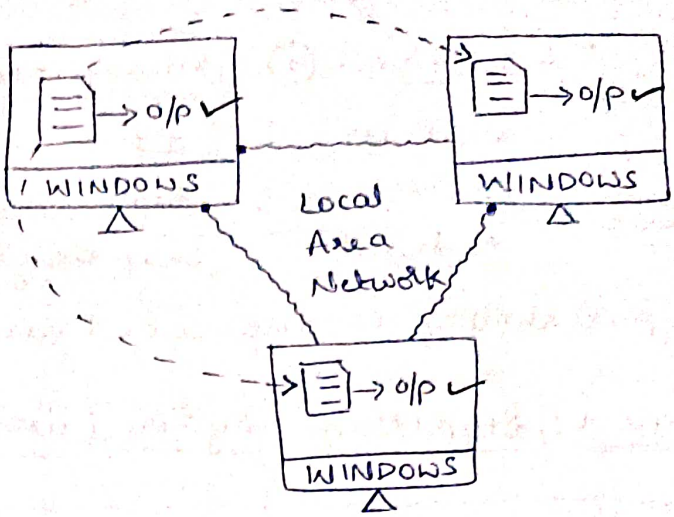
6/1/23

⇒ 1950's - 1960's



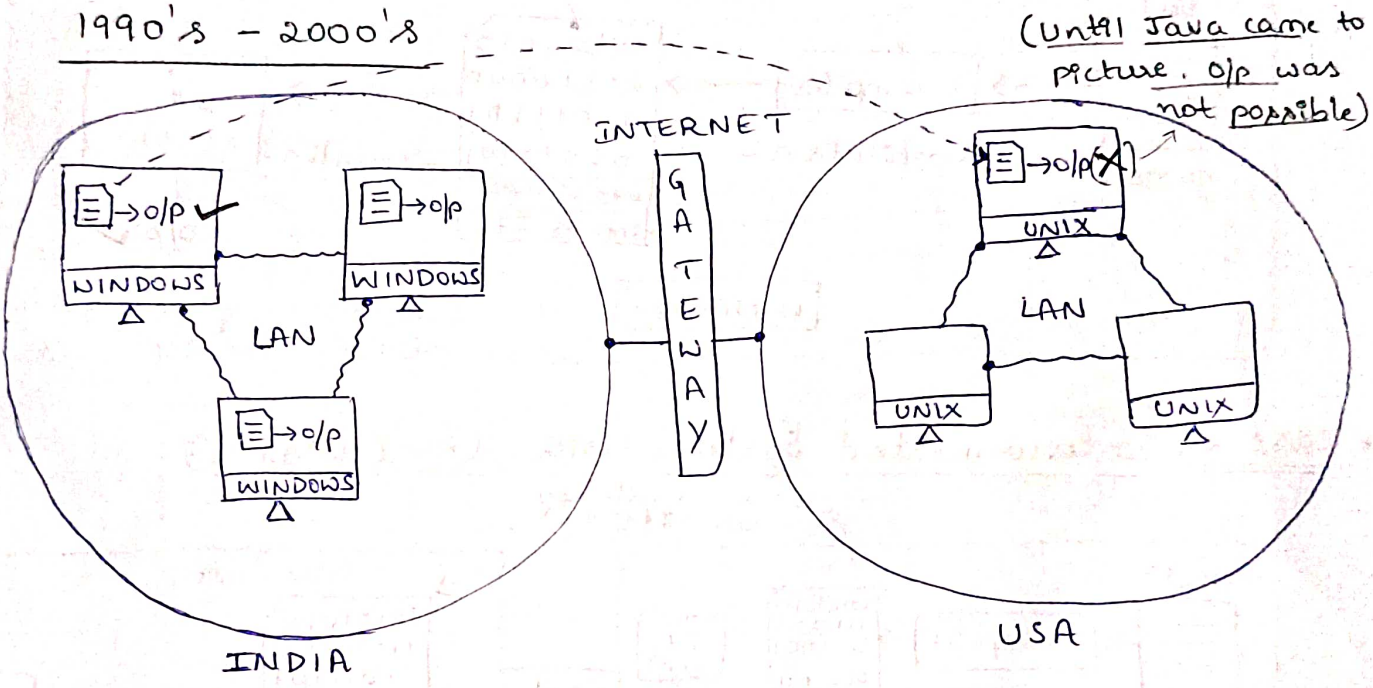
(Stand Alone)

1970's - 1980's (LAN)



1990's - 2000's

(Until Java came to picture, o/p was not possible)



- With the Introduction of Internet, the demand for portable languages saw a rise.
- programs written on one platform (configuration) failed to give o/p when executed on different platform

Portability:

- It is the capacity of a programming language to be compiled on one platform & executed on another platform.
- platform is a combination of OS (operating system) & µp (Microprocessor).

platform

OS ⊕ Microprocessor

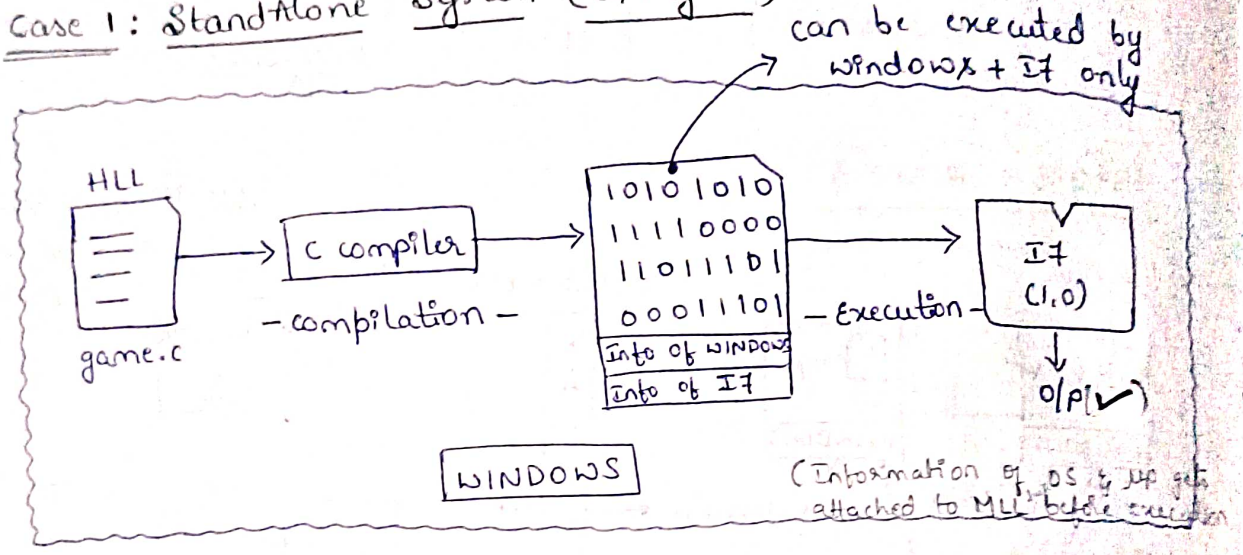
WINDOWS - I7

UNIX - AMD

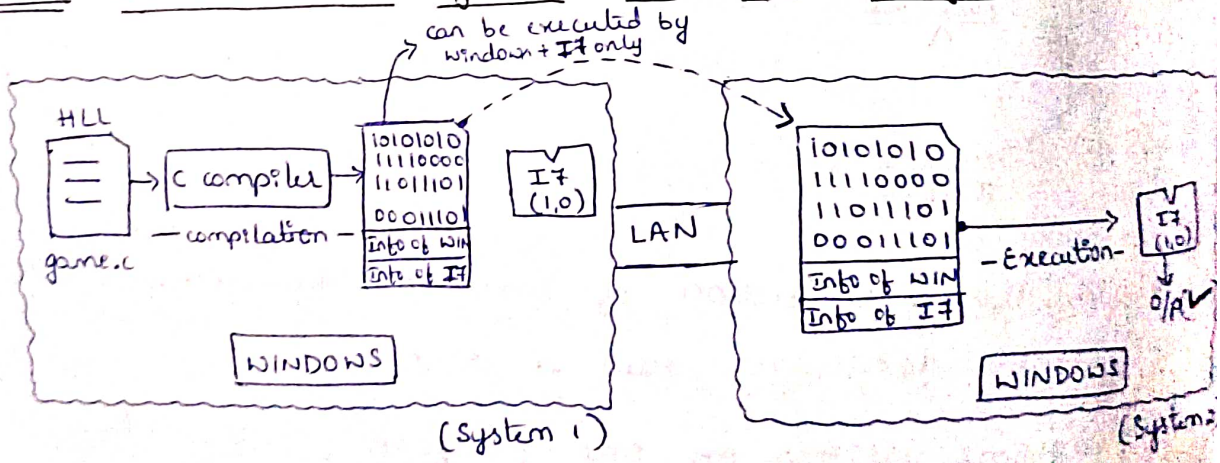
Android - Snap Dragon

- portability is also called as platform independent

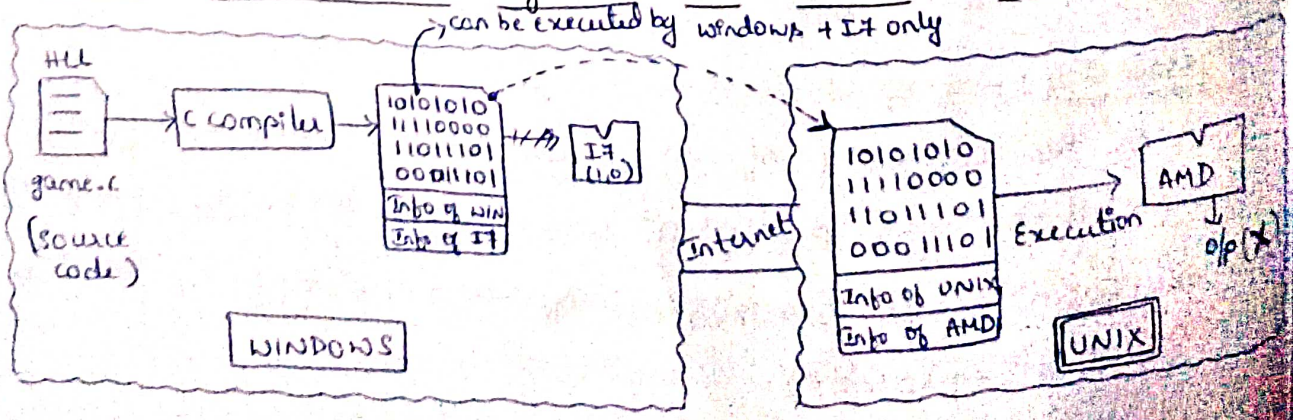
→ Case 1: StandAlone System (using c)



→ Case 2: Interconnected Systems with LAN (using c)

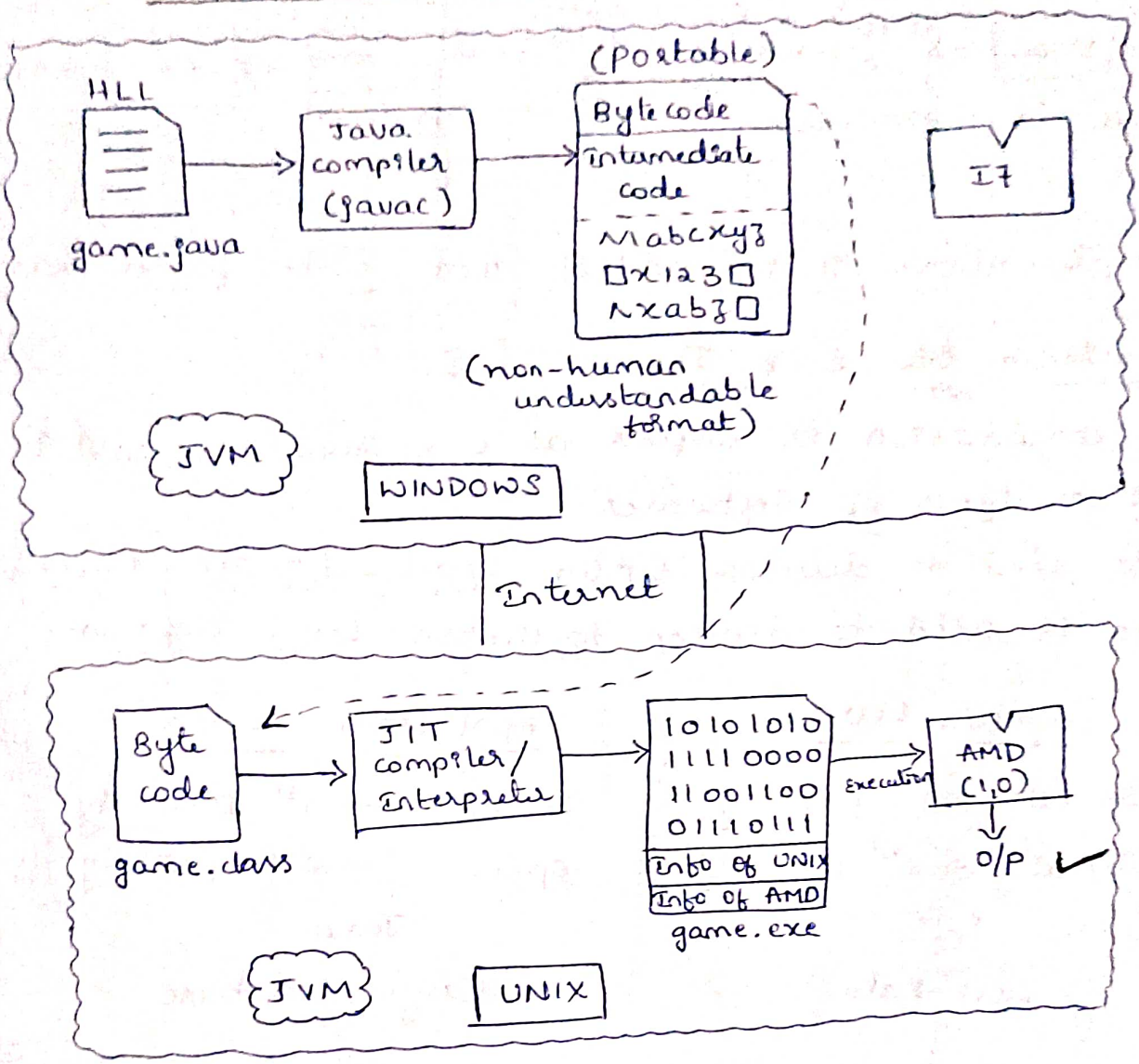


→ Case 3: Interconnected Systems with Internet (using c)



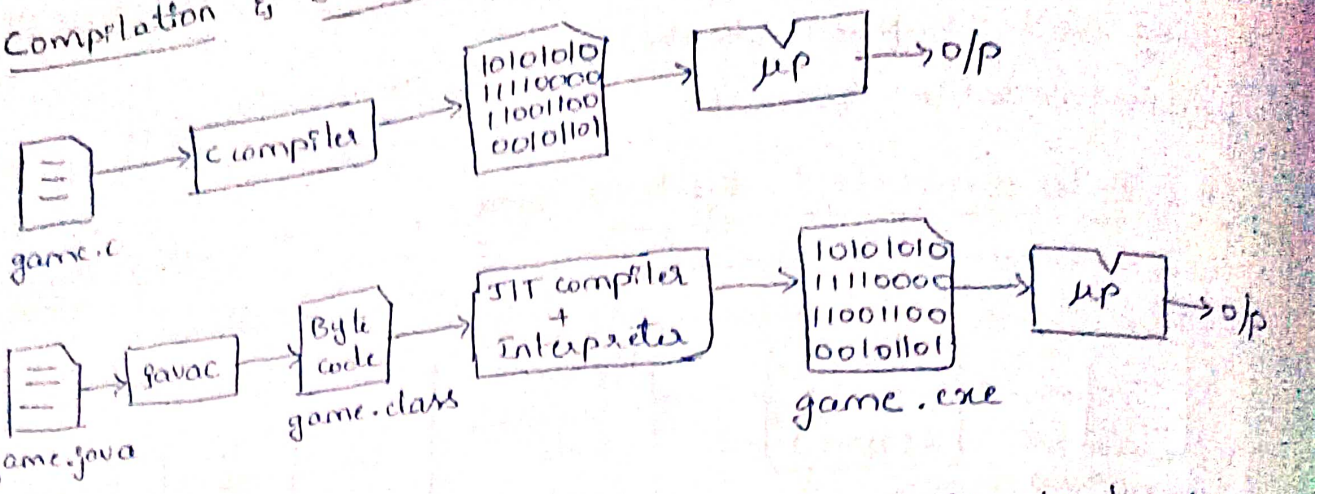
- Source code should always be converted to non-human understandable format before sharing.
- Binary code happens to be one of the non-human understandable format.

⇒ Case 4 : Interconnected systems with Internet (using Java)



- In case 4, game.java is converted into Bytecode after initial compilation by java compiler.
- game.java now becomes game.class.
- Bytecode is the portable intermediate code that contains the program in non-human understandable format.
- Just before execution bytecode is converted to executable binary code by JIT compiler & Interpreter.
- game.class now becomes game.exe
- In both the systems where the java pgm is compiled & executed a special software known as Java Virtual Machine (JVM) must be installed.

Compilation & Execution process in C v/s Java :



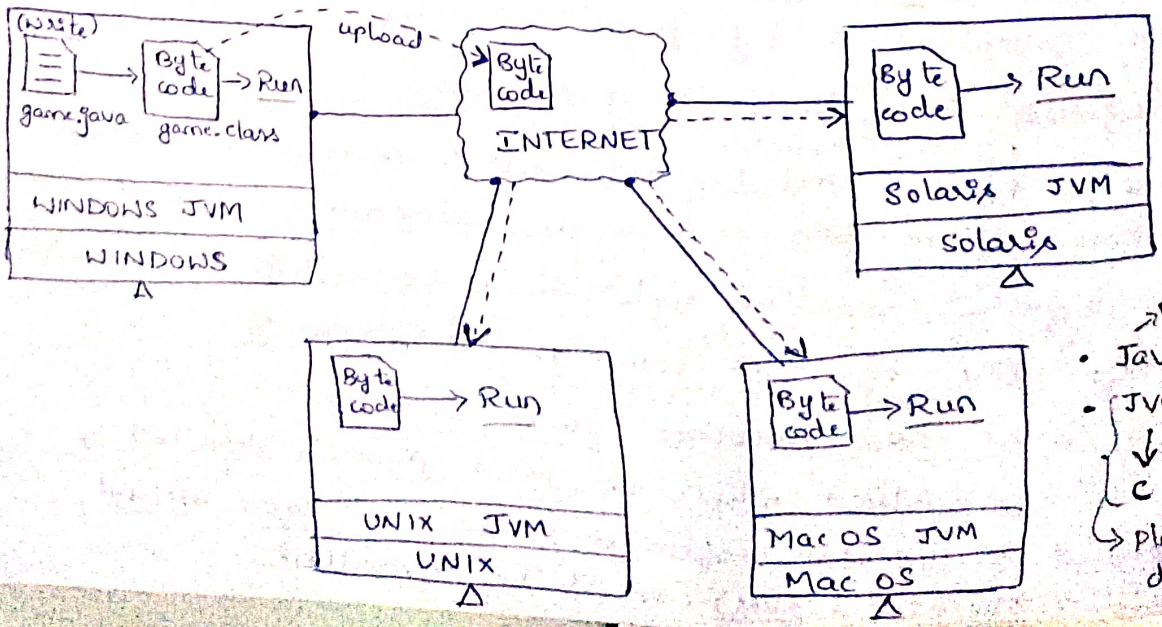
From the above it is evident that C is faster than Java.

Comparision b/w C & Java :

- This comparision is unfair as C & Java are used to build different types of softwares.
- C is used to develop System Level software whereas Java is used to develop Application Level software.

System Level	Application Level
* Speed → 1st priority	Security → 1st priority
* Security → 2nd priority	Speed → 2nd priority
* "C"	"Java"
* Less Data	Large Database

WORA (Write once Run Anywhere) :



→ Platform independent

- Java pgm → Pat
- JVM → Not pat
  - C (not portabl
  - platform dependent

- Java programs are portable but JVM is not.
- On a windows system a windows compatible JVM must be installed. On a Unix system a unix compatible JVM must be installed and so on.

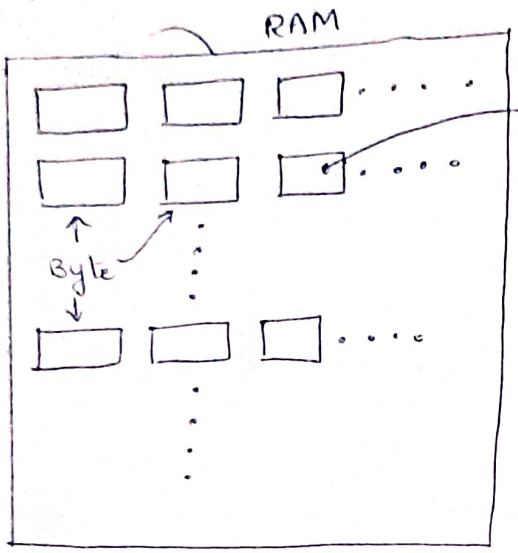
Organization of a Computer:

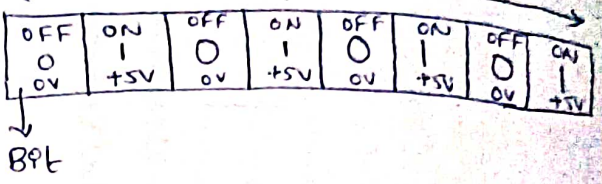
(Loading → the process of transferring the files/programs from Hard disk drive to RAM during execution.)

Note:

- ROM (Read only Memory) is also a part of primary memory but it is dedicated only for internal programs (system BIOS) of the computer.

Organization of RAM:





1 Bit = 1 FlipFlop = 2 NAND/NOR gates

4GB = 400,00,0,0,000 Bytes

= 3200,00,00,000 Bits (4x8)

= 32 00,00,00,000 FlipFlops

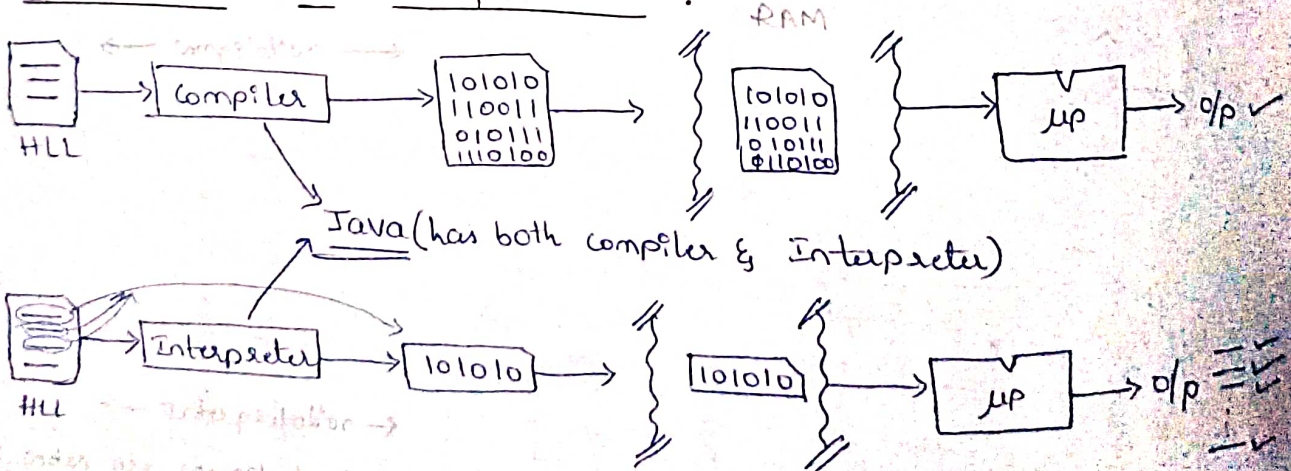
= 6400,00,00,000 units (VLSI) (devices)

Very large scale integration

4GB = 4 GigaBytes = 4×10^9 Bytes

- Random Access Memory is a part of primary memory.
- It is also called as Main memory of the computer.
- It is a collection of Bytes inside which we store the data.

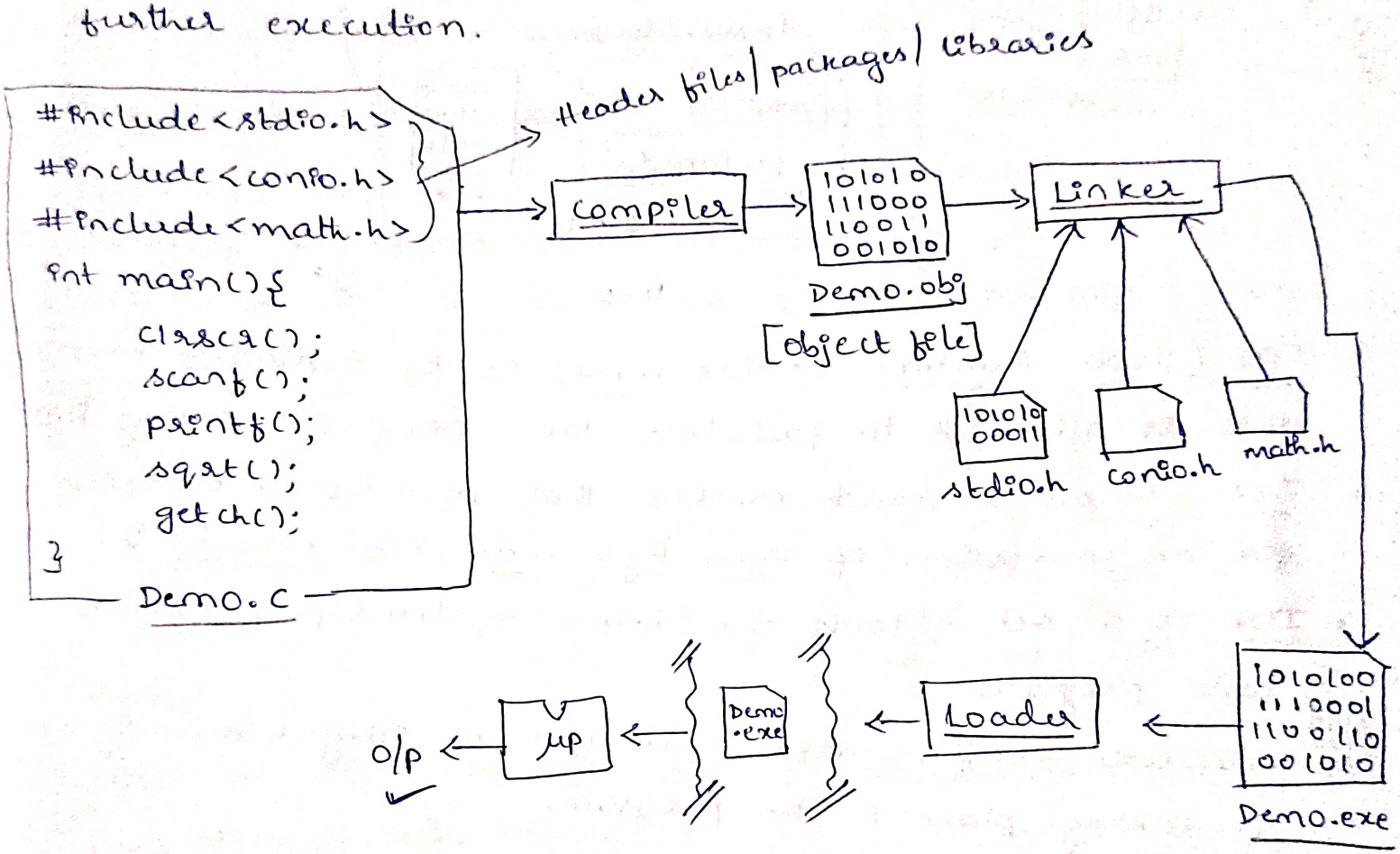
Compilation v/s Interpretation:



- Generally Interpreter is slower when compared to Compiler.
- Java has both compiler & Interpreter.
- Internally the program will be split, some parts will be given to compiler & some parts will be handled by interpreter. (multithreading)

Object file V/s Executable file :

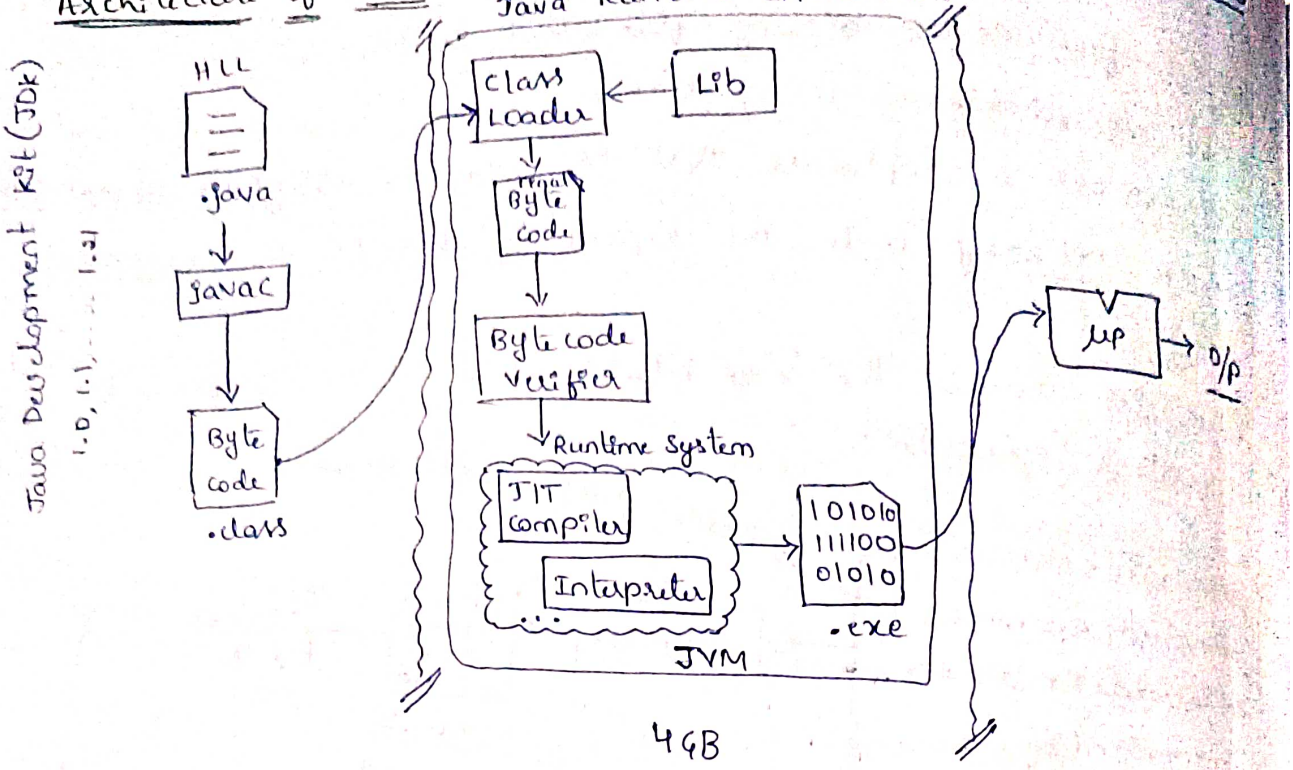
- Object file is a binary file that is incomplete & hence not executable.
- Linker is a software that is going to integrate object file with library code. The result of this is the Executable file.
- Loader loads the executable file on to the RAM for further execution.



Object file is incomplete because it contains the binary code of methods/statements like clrscr(), printf()... but it doesn't contain the library code like (stdio.h, conio.h)

8/7/23

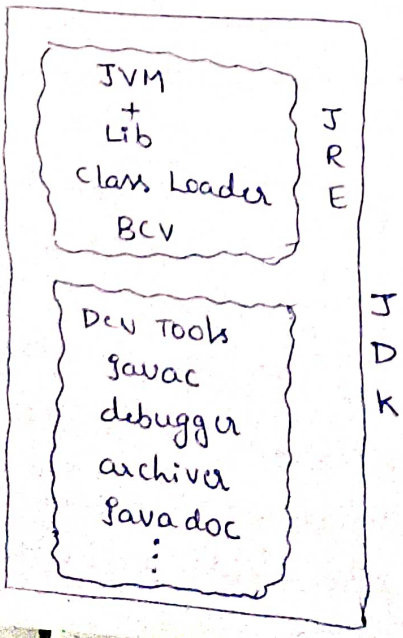
Architecture of Java :



- JRE (Java Runtime Environment) is the dedicated memory that is allocated to facilitate Java programs during Runtime
- JVM is an abstract machine that provides an environment for the conversion of Java Byte code into executable code.
- JDK is a set of tools that helps to develop & execute Java programs.

Note:

* Byte code verifier makes sure that no illegal (pointer) memory access is taking place in the program.
- class Loader loads the Java program on to the RAM & integrates it with the Library code.



Object Oriented Programming:

- It is a programming paradigm (methodology/ style/ approach) that provides solutions to problems keeping software objects as a base. & using object oriented concepts like Encapsulation, Inheritance, polymorphism & Abstraction. The above are called as pillars of object oriented programming.

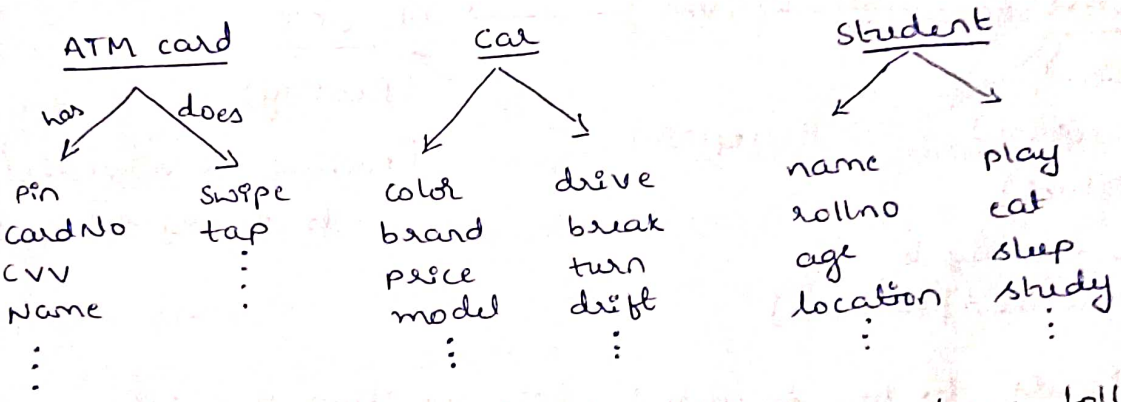
Object: It is an instance of a class.

- It contains data & methods that work on the data.

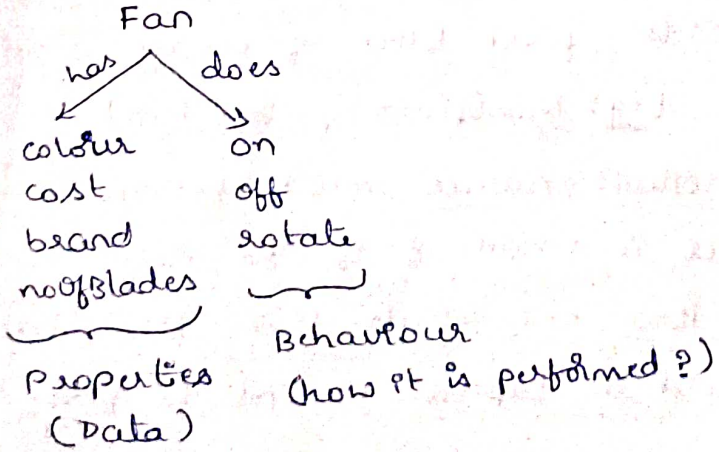
Steps involved in using objects to solve problems:

1. Identify the objects.
2. classify the objects based on type.
3. Define ~~the~~ what the object has & what the object does.

eg:



⇒ Create an object of type Fan. It should have following properties & behaviours.



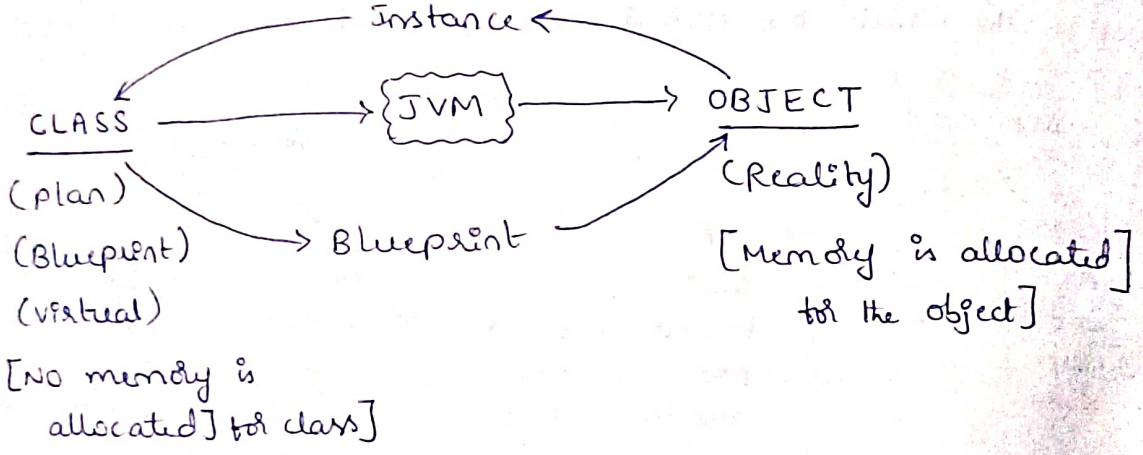
↳
```
class Fan{
    String colour = "white";
    int cost = 2000;
    String brand = "Havells";
    int noOfBlades = 4;
```

properties/ Instance variables/ Data members

```
void on(){
    System.out.print("fan is on");
}
void off(){
    System.out.print("fan is off");
}
void rotate(){
    System.out.print("fan is rotating");
}
}
```

Behavioural Methods

- Class is a blueprint for an Object.
- Inorder to create the Object, the class must be given to JVM



Standards to be followed while writing a class:

i) Class name should follow pascal case where first letter should be capital & if the class name is a combination of more than one words, first letter of every word should be capital. [eg: NumOfFans, class Fan]

ii) property names & method names must follow camel case where the first letter is small & if the name is a combination of more than one words then first letter of every word should be capital except the first word. [eg: noOfBlades]

iii) The class should be logically coherent.

Eg for Incoherent class:

```
class Demo {
    int x = 10;
    int y = 20;
    String z = "Yellow";
    void abc(){
    }
}
```

- To make sure that the class is coherent, class name & property names should be Noun's whereas method names should be Verbs.
- In order to create an object of the above Fan class we use the following command:

```
new Fan();
```

Primitive DataTypes in Java:

- Integer
  (1, 100, -100, 999, 0...)
  byte - 1
  short - 2
  int - 4
  long - 8
- Real Numbers
  (1.11, -33.33, 0.469, ...)
  float - 4
  double - 8
- character
  ('a', '1', '$', ' ', 'abc' ×)
  char 2
- Boolean
  (true, false)
  boolean → JVM dependent

Needs to Specify data type:

i) To indicate amount of memory to be allocated.

ii) To indicate nature of the data.

Note: character is a single alphabet (or) number (or) symbol within single quotations.

Non-primitive Data Types: class, Interface, String, Arrays...

Variables / Identifiers:

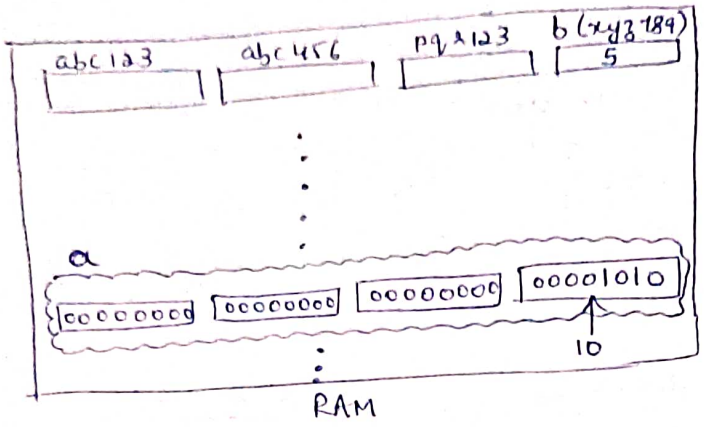
- It is the name given to memory location within which the data is stored.
- It acts as a container to hold data.

eg: i) int a = 10; (I want the data 10 to be stored. It is of type 'int'. Allocate memory for it & name the location as 'a').
(Initialization)

ii) byte b = 5; (I want the data 5 to be stored. It is of type 'byte'. Allocate memory for it & name the location as 'b').

iii) int c; (I want to store data of type 'int'. Allocate memory for it & name the location as 'c'. Data will be provided later.

↑
(Declaration)



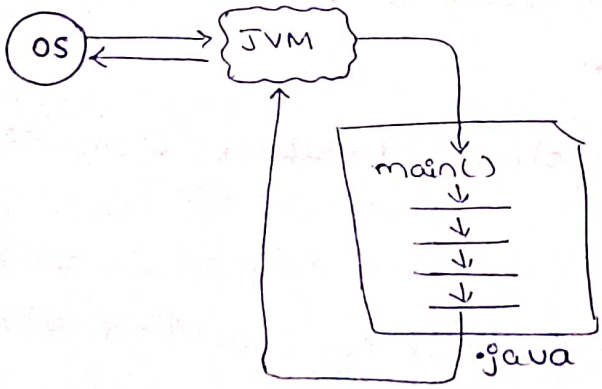
Note:

5 → 0 0 0 0 0 1 0 1
2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
128 64 32 16 8 4 2 1

10 → 0 0 0 0 1 0 1 0
8 4 2 1

Control Flow:

- Ultimate control of the system will be with operating system that coordinates all the activities of the computer.
- When a program has to be executed OS transfers the control to the program.
- The path the control traces throughout the execution of the program is called as control flow.



Note:

Rules for naming variables:

i) No spaces allowed in the name.

ii) Cannot start with a number (or) special symbol except underscore (_)

iii) Should follow camel case.

Assignment Operator (=):

- Assignment Operator is used to evaluate the RHS & return the result to the LHS. Hence a statement containing '=' should be read from Right to left.
- eg: a = 10 ; (10 is given to a)

In maths	In programming
a=5 ✓	a=5; ✓
5=a ✓	5=a; ✗

- Variable = Expression
  (updated) (Evaluated)
  ↓ storage

(variable on left side of equal to is always updated.)

Steps involved in Object creation:

i) Create an address variable.
ii) Using 'new' allocate memory for the object on the RAM.
iii) Allocate memory for instance variables inside the object.
iv) Collect the address of the object. returned by new. using address variable.

The two jobs of new:

Job 1. Allocate memory as requested by the programmer.
Job 2. Return the address of allocated memory.

=> Write a student class & create an object of it.

StudentApp.java

```
class Student {
    String name = "Arjun";
    int age = 23;
    int marks = 45;
    void study() {
        s.o.p("student is studying");
    }
}
```

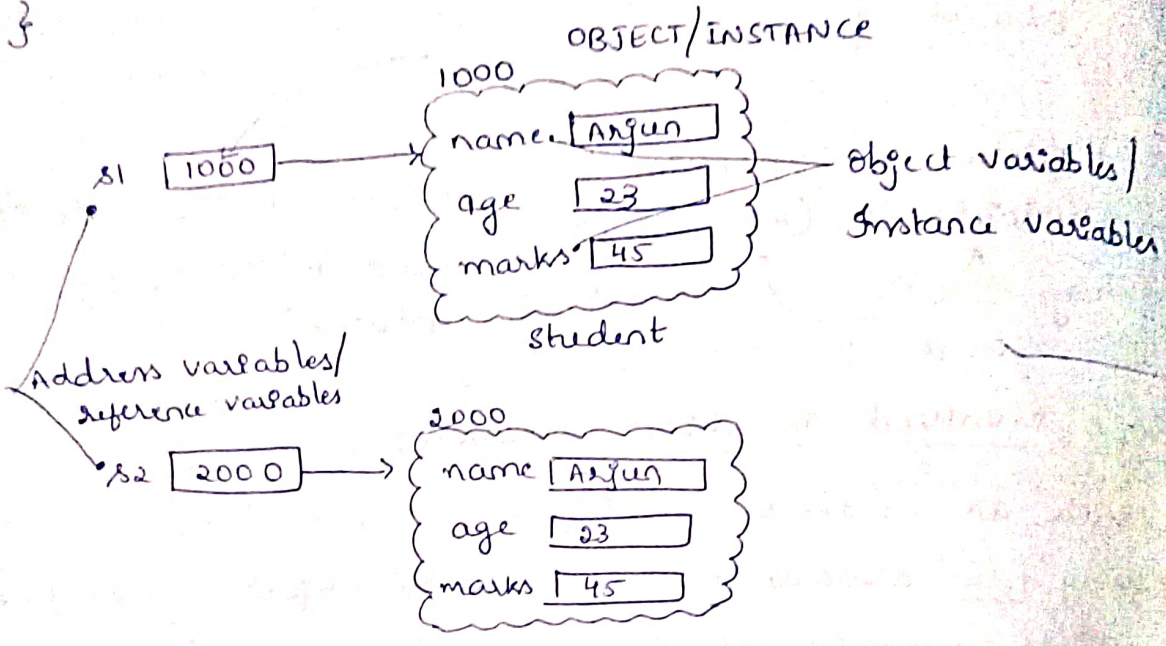
```
class StudentApp{
    public static void main(String args[]){
        Student s1;
        s1 = new Student();
```

1000

```
        Student s2;
        s2 = new student();
```

2000

```
    }
}
```



Segments of RAM:

CODE SEGMENT	→ program
STATIC SEGMENT	→ static variables → static block → static methods
STACK SEGMENT	→ Activation records with local variable
Heap SEGMENT	→ Objects

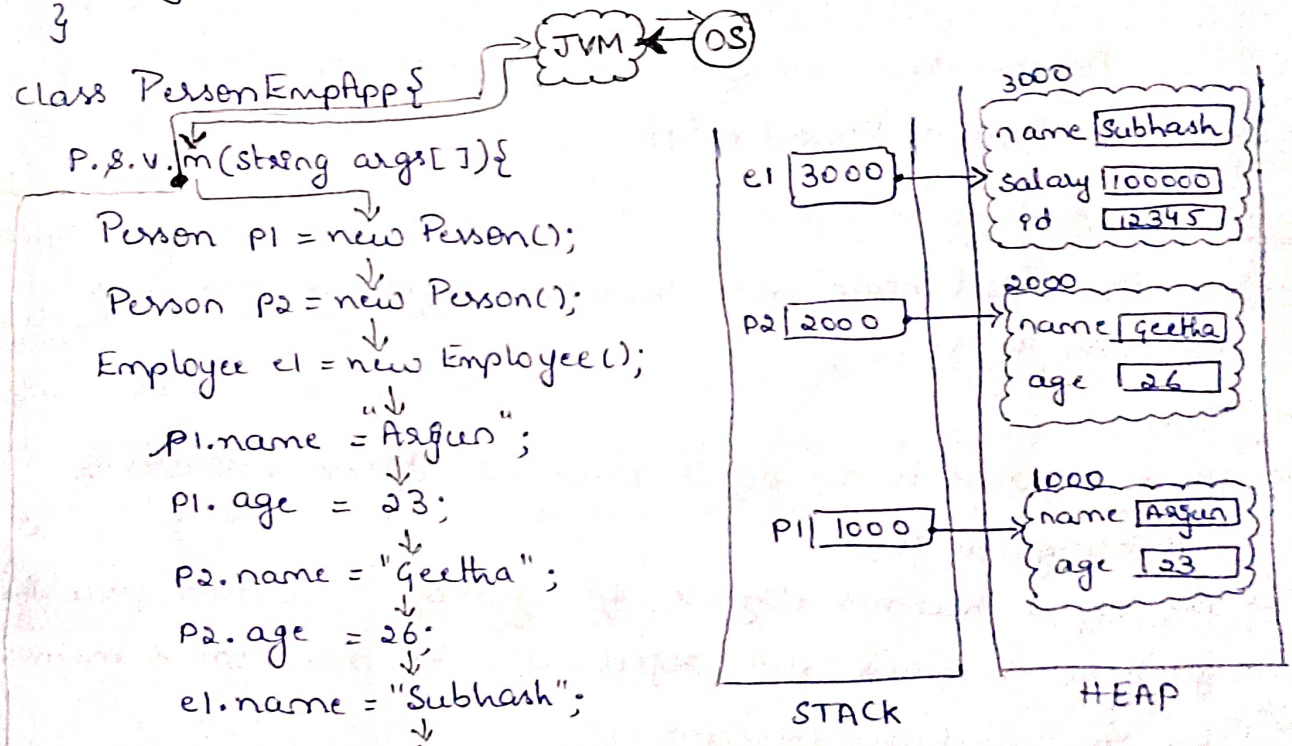
Local variables are those variables which are declared within methods.

```
→ eg: P S V m(){
        int a;
        int b;
        float c;
    }
}
```

→ Local variables

Eg for Object Creation:

```
class Person{
    String name;
    int age;
    void play(){
        S.o.p("Person is playing");
    }
}
class Employee{
    String name;
    int salary;
    String id;
    void work(){
        S.o.p("Employee is working");
    }
}
class PersonEmpApp{
    P.S.V.m(String args[]){
        Person p1 = new Person();
        Person p2 = new Person();
        Employee e1 = new Employee();
        p1.name = "Arjun";
        p1.age = 23;
        p2.name = "Geetha";
        p2.age = 26;
        e1.name = "Subhash";
        e1.salary = 100000;
        e1.id = "12345";
        S.o.pln(p1.name);   // Arjun
        S.o.pln(p1.age);    // 23
        S.o.pln(p2.name);   // Geetha
        S.o.pln(p2.age);    // 26
        S.o.pln(e1.name);   // Subhash
        S.o.pln(e1.salary); // 10000
        S.o.pln(e1.id);     // 12345
    }
}
```



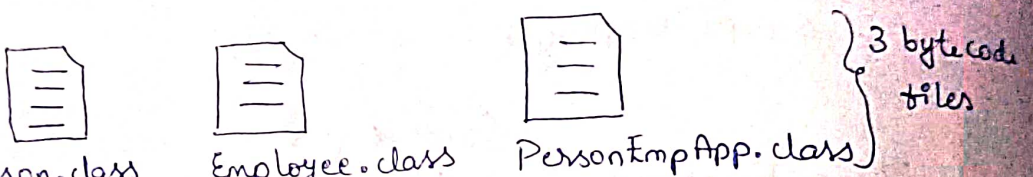
- The above program should be called as

PersonEmpApp.java

- Any no of classes can be written in a single program & any no of objects can be created for each class.
- When the above program is compiled, it generates 3 bytecode files as the program contains 3 classes.
- command to compile a java program → javac filename
- command to execute a java pgm → java filename.
- For the above pgm, the commands are as follows:

⇒ c:\...>javac PersonEmpApp.java

⇒



⇒ c:\...>java PersonEmpApp

Note:

Q1. Can one object have more than one reference variables pointing to it?

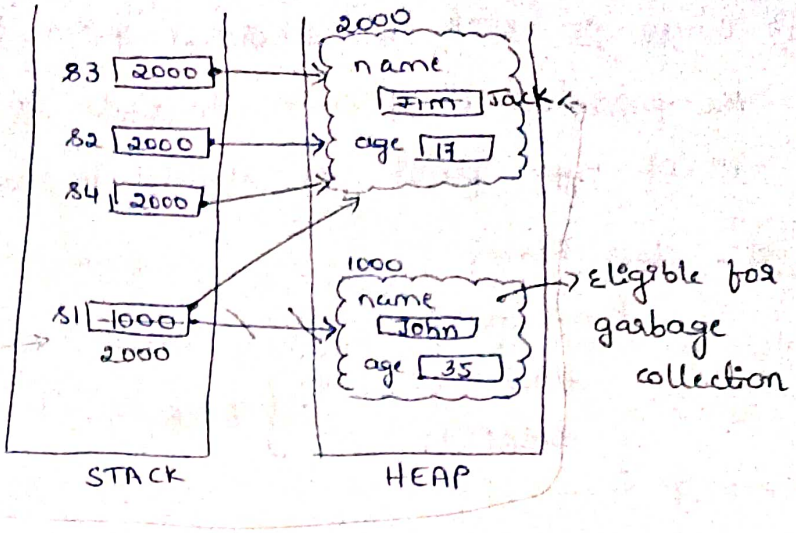
↳ Yes.

Q2. What happens to an object when no address variables is pointing to it?

↳ The object becomes eligible for garbage collection. Once the garbage becomes full object will be deallocated memory by the garbage collector.

eg:
```
class Student{
    String name;
    int age;
}
class StudentApp{
    P.S.V.M(){
        Student s1, s2, s3, s4;
        s1 = new Student();
        s2 = new Student();
```

```
S1.name = "John";
S1.age = 35
S2.name = "Jim";
S2.age = 17;
S3 = S2;
S4 = S2;
S1 = S2
S3.name = "Jack";
```




```
S.o.p (S1.name); // Jack
S.o.p (S2.name); // Jack
S.o.p (S3.name); // Jack
S.o.p (S4.name); // Jack
    }
}
```

- In the above pgm, multiple address variables are pointing to the same object.
- Here we must be careful that any changes made to the object w.r.t one variable will reflect w.r.t all address variables.

Note: 10/7/23

;) The default value of local variables is garbage. Hence local variables should not be used in an expression (or) should not be printed before initializing. This is bcz garbage values are not allowed in Java & using them will lead to an error.

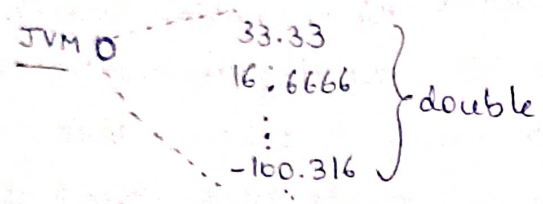
eg:
```
class Demo{
    P.S.V.m(){
        int a;
        int b=10;
        int c=20;
        S.o.p(a);      // Error
        S.o.p(b);      // 10
        S.o.p(c);      // 20
        b=c+a;         // Error (20+G=G)
        a=b+c;         // 30 (10+20)
        S.o.p(a);      // 30
    }
}
```

a ~~G~~ 30
b ~~G~~ 10
c ~~G~~ 20

ii) Every no with a decimal point is treated as double from the perspective of JVM. In order to specify that the value is of type float, it should be suffixed with letter 'f'.

eg:
```
class Demo{
    P.s.v.m(){
        float c = 33.33;   } Error
        S.op(c);
    }
}
```

```
class Demo{
    P.s.v.m(){
        float = 33.33f;
        S.o.p(c);   // 33.33
    }
}
```



=> Write a program to find the sum of 2 numbers.

```
-> class SumOfTwoNum{
    P.s.v.m(){
        int a=10, b=20;
        int sum;
        sum = a+b;
        S.o.p(sum);   //30
    }
}
```

← The above pgm is hard coded i.e the programmer themselves have provided the data in the pgm. This is not an automated soln as the pgm needs to be changed each time the sum of two different no's has to be found. The pgm should be capable of taking input from user & provide output for any data given. Hence making it automated

-> Note:

```
int x = 100;
S.o.p(x);      -> // 100
S.o.p("x");    -> // x
```

System.out.print(....);
- System: class
- out: variable
- print(....): method

Anything written within " " will be printed as it is

S.o.p("sum of a and b = sum");
↓
o/p: sum of a and b = sum

→ Expected Output:

Enter the first number : 10 ↵
Enter the second number : 20 ↵
Sum of 10 and 20 = 30

In order to scan the input given by the user & pull it into the program we must take help of methods given inside the Scanner class. This scanner class is present inside a package called util which in turn will be present in a package called java.

java → util →

```
class Scanner{
    nextInt(){          // scan 'int' type data from
      ---                  keyboard.
    }
    nextFloat(){        // scan 'float' type data.
      ---
    }
    next(){             // scan 'string' type data with
      ---                  no spaces.
    }
    nextLine(){         // scan 'string' type data
      ---                  with/without spaces.
    }
    nextDouble(){       // scan 'double' type data.
      ---
    }
}
```

↳ PE import java.util.Scanner;

```
class SumOfTwoNum {
    P.S.V.m(){
        int a, b, sum;
        Scanner sc = new Scanner(System.in);
```

```
System.out.println("Enter the first number:");
    a = sc.nextInt();
          (10)
S.o.pln("Enter the second number:");
    b = sc.nextInt();
          (20)
    sum = a+b;   (10+20=30)
S.o.p("Sum of "+ a +" and "+ b +" = "+ sum);
  }
}
```

a → 10, b → 20, c → 30

Concatenation of strings

=> Write a program to find the square of a given number.

```
import java.util.Scanner;
class SquareOfNum{
    P.S.V.m(){
        int a, sqr;
        Scanner sc = new Scanner(System.in);
        S.o.p("Enter the value:");
        a = sc.nextInt();
              (11)
        sqr = a*a;
              "*" = 121
        S.o.p("Square of "+a+" = "+sqr);
    }
}
```

Expected output:

```
Enter the value: 11 ↵
Square of 11 = 121
```

Operators: These are symbols that perform predefined operations.

Types of Operators in Java:

1) Assignment Operator (=)

ii) Arithematic Operators : (+, -, *, /, %, ++, --)
(Binary) (B) (B) (B) (Unary) (U)

→ / (Normal division → Quotient)

In maths	In Java
15/4 = 3.75	15/4 = 3
33/3 = 11	33/3 = 11
16/3 = 5.3333	16/3 = 5

Int * Int = Int

100 % 20

5 → Quotient

20) 100

100

0 → Remainder

→ % (Remainder division → Remainder)

15 % 4 = 3

33 % 3 = 0

16 % 3 = 1

100 % 99 = 1

99 % 100 = 99

→ ++ (increase value by 1) ; -- (decrease value by 1)

```
Int i = 10;
i++;
i++;
S.O.p(i);    // 12
i--;
S.O.p(i);    // 11
```

iii) Comparision Operator : (<, >, <=, >=, ==, !=)
(B) (B) (B) (B) (B) (B)

a = 10

b = 20

(a > b) → False (boolean)

(a < b) → True

(a >= b) → F

(a <= b) → T

(a != b) → T

(a == b) → F

iv) Logical operators: (|| (OR), && (AND), ! (NOT))

|| (OR)

C1	C2	Result
T	T	T
T	F	T
F	T	T
F	F	F

&& (AND)

C1	C2	Result
T	T	T
T	F	F
F	T	F
F	F	F

! NOT

```
boolean status = true;
S.O.P(!status); //false
```

v) Tanary operator: (? :) (Ternary)

variable = (condition) ? value1 : value2

True

False

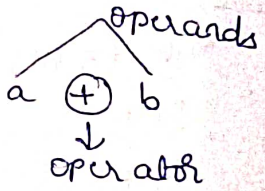
vi) Bitwise operators

Note: Based on no of operands there are 3 categories of operators:

a) Unary → 1 operand (U)

b) Binary → 2 operands (B)

c) Ternary → 3 operands (T)



Selection statements:

- These statements help us to decide on the execution of a set of statements based on the result of the given condition.

```
→ if(condition){
      S1;
      S2;
  }
  else{
      S3;
      S4;
  }
```

If the condition is true, we execute these statements

Executed if the condition is false

if - else if Ladder:

```
if(condition 1){
}
```

↕ C1 → T

```
else if (condition 2){          ↕ C1 → F
                                  C2 → T
}
else if (condition 3){          ↕ C1 → F
                                  C2 → F
                                  C3 → T
}
else if (condition 4){          ↕ C1, C2, C3, → F
                                  C4 → T
}
else{                           ↕ All conditions are false
}
```

* Out of the above set of statements, only one will be selected for execution. Once one of the conditions becomes true, the rest of the will not be checked.

=> Write a pgm to check if a Student has passed (or) failed based on the marks. (pass marks = 35)

```
import java.util.Scanner;
class PassFail{
    P.S.V.M(){
        Scanner sc = new Scanner(System.in);
        S.o.p ("Enter the marks :");
        int marks = sc.nextInt();
        if (marks >= 35){
            S.o.p("congratulations!!! Pass");
        }
        else{
            S.o.p ("Sorry !!! Fail");
        }
    }
}
```

O/P: i) Enter the marks : 46 ↵
congratulations!!! Pass

ii) Enter the marks : 33 ↵
Sorry!!! Fail

=> Write a pgm to check Eligibility for voting using Ternary Operator.

```
import java.util.Scanner;
class Voting{
    p.s.v.m(){
        int age;
        Scanner sc = new Scanner(System.in);
        S.o.p("Enter age:");
        age = sc.nextInt();
        String res = (age>=18)? "Eligible": "Not eligible";
        S.o.p(res);
    }
}
```

- O/P: i) Enter age : 27 ↵
  Eligible

  ii) Enter age : 12 ↵
  Not Eligible

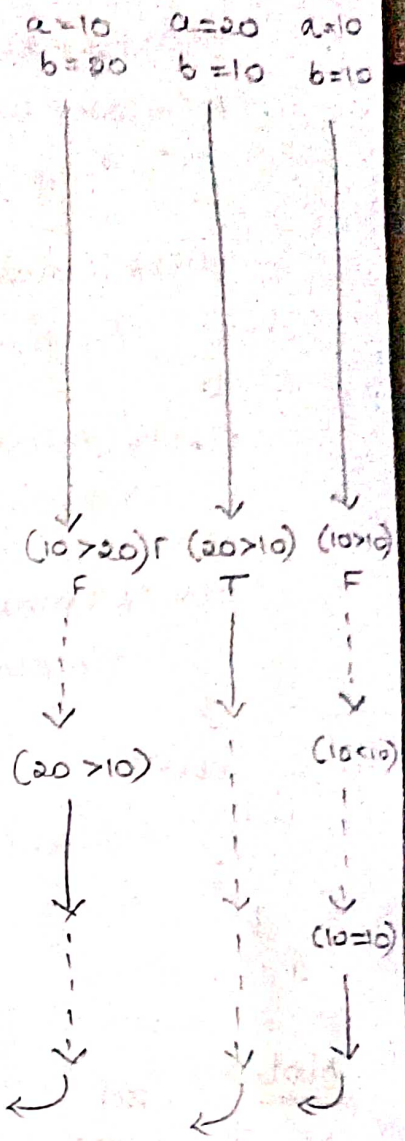
=> Write a pgm to check the greatest among two numbers.

- Expected o/p: i) Enter value of a : 10 ↵
  Enter value of b: 20 ↵
  b is greater

  ii) Enter value of a : 20 ↵
  Enter value of b : 10 ↵
  a is greater

  iii) Enter value of a : 10 ↵
  Enter value of b : 10 ↵
  a and b are equal

```
import java.util.Scanner;
class GreatestOfTwoNum{
    P.s.v.m(){
        Scanner sc = new Scanner(System.in);
        int a, b;
        S.o.p("Enter value of a:");
        a = sc.nextInt();
        S.o.p("Enter value of b:");
        b = sc.nextInt();
        if(a > b){
            S.o.p("a is greater");
        }
        elseif(b > a){
            S.o.p("b is greater");
        }
        else{
            S.o.p("a and b are equal");
        }
    }
}
```



11/7/22

⇒ Write a pgm to assign grades based on the marks.

- Grades — marks

Grades	marks
A	90 and above
B	75+ & upto 89
C	60+ upto 74
D	35+ upto 59
Fail	Less than 35

```
import java.util.Scanner;
class Grades{
    P.s.v.m(){
        Scanner sc = new Scanner(System.in);
        int marks;
```

```
S.o.pln("Enter the marks:");
marks = sc.nextInt();
if(marks >=90){
    S.o.pln ("Grade A");
}
elseif (marks >=75 && marks <=89){
    S.o.pln ("Grade B");
}
else if(marks >= 60 && marks <=74){
    S.o.pln ("Grade C");
}
else if (marks >=35 && marks <=59){
    S.o.pln ("Grade D");
}
else{
    S.o.pln("Fail");
}
}
}
```

marks = 43

↓

(43 ≥ 90) (F)

↓

(43 >= 75) && (43<=89)

F && T

↓

(43 >= 60) && (43<=74)

F && T = (F)

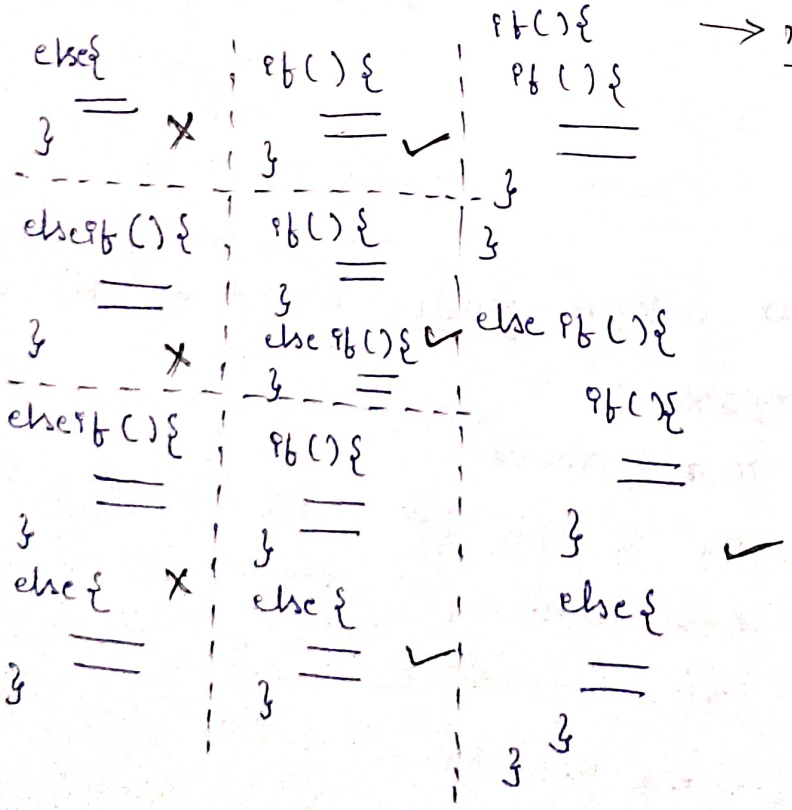
↓

(43 >= 35) && (43<=59)

T && T = (T)

↓

Note:



→ Nested if-else

Switch Statement:

Syntax:

```
switch (Variable Name){
    case value1 : s1;
                  s2;
                  :
                  break;
    case value2 : s1;
                  s2;
                  :
                  break;
    :
    case valuen : s1;
                  s2;
                  :
                  break;
    default : s1;
              s2;
              :
}
```

- The variable name provided within switch can be of any type. The value of the variable is compared with the values provided in the cases one by one. If a match is found then the corresponding statements are executed. If no match is found then statements corresponding to default are executed.
- Break statement is optional. It is provided to ensure the comparision doesn't continue even after finding a match.

⇒ Write a pgm to display the day name based on the day number.

- Expected o/p: i) Enter day number : 5 ↵
  Thursday
  ii) Enter day number : 1 ↵
  Sunday

```
import java.util.Scanner;
class WeekDay{
    P.S.V.m(){
        Scanner sc = new Scanner(System.in);
        int day;
        S.o.p("Enter day number:");
        day = sc.nextInt();
        switch(day){
            case 1 : S.o.pln("Sunday");
                     break;
            case 2 : S.o.pln("Monday");
                     break;
            case 3 : S.o.pln("Tuesday");
                     break;
            case 4 : S.o.pln("Wednesday");
                     break;
            case 5 : S.o.pln("Thursday");
                     break;
            case 6 : S.o.pln("Friday");
                     break;
            case 7 : S.o.pln("Saturday");
                     break;
            default : S.o.pln("Invalid Input");
        }
    }
}
```

Looping Statements:

- When a set of statements need to be executed repeatedly then we can take the help of looping statements.
- By placing statements within a loop, we can execute them again & again until a condition is matched.

⇒ Write a pgm to print your name on screen 5 times.

```
class LoopIntro {
    P.S.V.m() {
        S.o.pln("Pentagon");
        S.o.pln("Pentagon");
        S.o.pln("Pentagon");
        S.o.pln("Pentagon");
        S.o.pln("Pentagon");
    }
}
```

} without looping

Types of Loops:

1) while Loop:

Syntax: while (condition){
    S1;
    S2;
    :
}

- The statements placed within while loop will be executed repeatedly until the condition becomes false.
- while using loops, we have to keep in mind 3 things
  @ where to start. ⓑ where to stop
  © How many steps to take.
- It is the responsibility of the programmer to provide a loop condition that becomes false at some point. Otherwise we will end up with an infinite loop.

↳

```
class WhileLoopIntro{
    P.S.V.m(){
        int i=1;   → (start)
        while (i<=5){   → (stop)
            S.o.pln("Pentagon");
            i++;
        }   → steps
    }
}
```

i: 1 2 3 4 5 6

I_1 (1<=5) → T
"pentagon" → ①
i++ ⇒ i=i+1 → i=1+1=2

I_2 (2<=5) T
"pentagon" → ②
i++ → i=i+1 → i=2+1=3

Iteration

I_3 (3<=5) T

"Pentagon" → ③

i++ → i=3+1=4

I_4 (4<=5) T

"Pentagon" → ④

i++ = i=4+1=5

I_5 (5<=5) T

"Pentagon" → ⑤

i++ = i=5+1=6

I_6 (6<=5) False

× stops looping

ii) For Loop:

Syntax:

```
for(Initialization ; Condition ; Increment/Decrement)
{
    S1;
    S2;
    :
}
```

Order of execution of for-loop:

```
for(Initialization ; Condition ; Inc/Dec)
{
    Statement 1;
    Statement 2;   } ③
    :
}
```

① ② ③ ④ — ② ③ ④ — ② ③ ④ — ② ③ ④ — ⋮ — ② → False

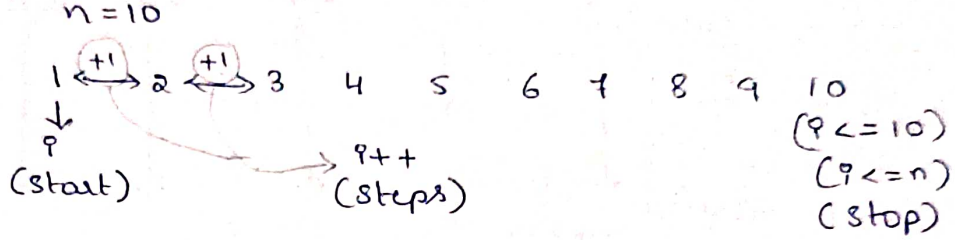
- Initialization happens only once.
- After that condition will be checked & if it is true, statements will be executed. Post this increment/decrement will take place. The above 3 steps will continue until the condition becomes false.

⇒ write a pgm to print first n natural numbers.

- Expected o/p: i) Enter value of n : 10↵
  The first 10 natural numbers are:
  1 2 3 4 5 6 7 8 9 10

  ii)
  1
  2
  3
  4
  5
  6
  7
  8
  9
  10

- Logic :



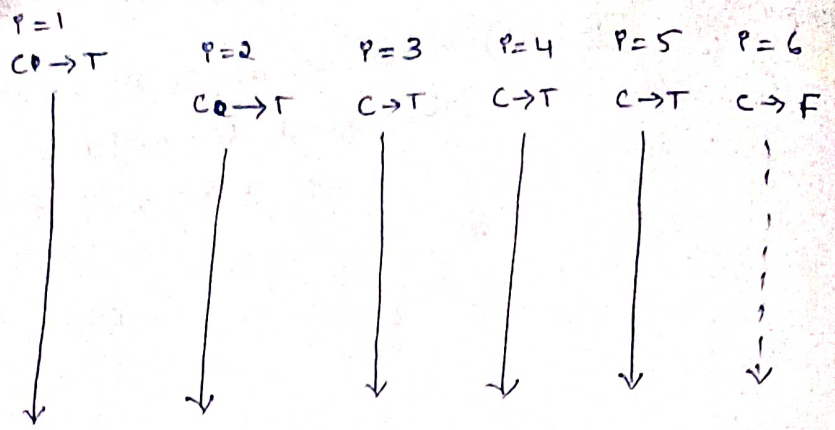

```
import java.util.Scanner;
class NaturalNum{
    P.S.V.M(){
      Scanner sc=new Scanner(System.in);
      int n, i;
      S.o.pln("Enter value of n:");
      n = sc.nextInt();
      S.o.pln("The first "+n+" natural numbers are:");
      for(i=1 ; i<=n ; i++)
      {
        S.o.p(i+" ");
      }                                   } Horizontal o/p
(OR)  for(i=1 ; i<=n ;i++)
      {
        S.o.pln(i);
      }                                   } vertically
    }
}
```

For loop → use for counting
while loop → use when we wait for something to happen.

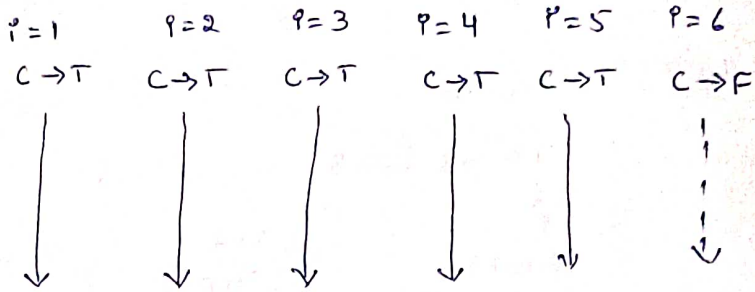
12/7/23

Note:

a)
```
Int i=1;
while (i<=5){
    S1;
    S2;
    :
    Sn;
    i++;
}
```
→



b)
```
for(i=1; i<=5; i++)
{
    S1;
    S2;
    :
    Sn;
}
```
→



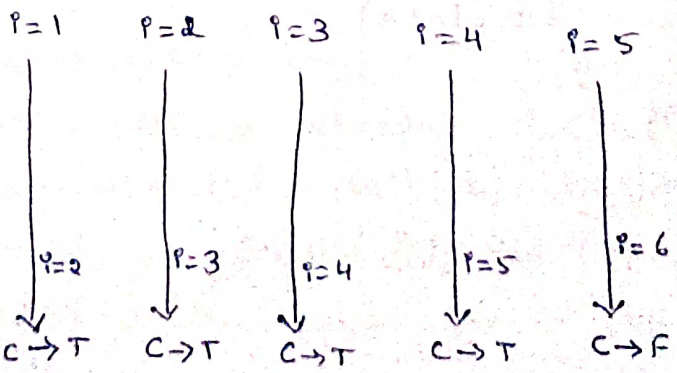
c) Infinite loop: A loop that never ceases its execution is called as infinite loop.

- It occurs when the condition governing the loop never becomes false. Hence it is the responsibility of the programmer to ensure that the condition becomes false at some point.

iii) do-while loop:

Syntax:
```
do{
    S1;
    S2;
    :
    Sn;
    i++;
}while(condition);
```

```
Int i=1;
do{
    S1;
    S2;
    :
    Sn;
    i++
}while(i<=5);
```



⇒ write a pgm to generate first n-even no's & first n-odd no's.

- Expected o/p: Enter Value of n: 10 ↵

First 10 even numbers are:

2 4 6 8 10 12 14 16 18 20

First 10 odd numbers are:

1 3 5 7 9 11 13 15 17 19

- Logic:

Even: n=10

2 <+2> 4 <+2> 6 20

(p) (start) p=p+2 (steps) p<=20 p<=2*n (stop)

odd: n=10

1 <+2> 3 <+2> 5 19

p (start) p=p+2 (steps) p<=19 p<=(2*n)-1 (stop)

If n=15,

First 15 even numbers are:

2 4 6 8 10 12 14 16 18 20 22 24 26 28 30

p (start) p=p+2 2n = 2x15 = 30 p<=2n (stop)

- import java.util.Scanner;

class EvenOdd {

P.S.V.m() {

Scanner sc = new Scanner(System.in);

Int n, p;

S.o.p("Enter Value of n:");

n = sc.nextInt();

S.o.pln("First " + n + " even numbers are:");

for(p=2; p<=2*n; p=p+2)

{

S.o.p(p+" ");

}

S.o.pln();

S.o.pln("First" + n + " odd numbers are:");

for(p=1; p<=2*n-1; p=p+2)

{ S.o.p(p+" ");

}

} }

⇒ Write a pgm to find Sum & product of the first n-natural nos.

- Expected o/p: Enter value of n: 6 ↵
  Sum of 1st 6 natural numbers are = 21
  Product of 1st 6 natural numbers = 720

- logic:
  21 → 1 + 2 + 3 + 4 + 5 + 6
  start (i=1)s (i++) steps (i<=6) stop
  720 → 1 * 2 * 3 * 4 * 5 * 6

```
import java.util.Scanner;
class SumProd{
P.S.V.m(){
Scanner sc = new Scanner(System.in);
int i, n, sum=0, prod=1;
S.o.pln("enter value of n: ");
n = sc.nextInt();
for(i=1; i<=n; i++)
{
sum = sum + i;
prod = prod * i;
}
S.o.pln("Sum of 1st" + n + " natural numbers =" + sum);
S.o.pln("Product of 1st" + n + "natural numbers =" + prod);
}
}
```

- Tracing: n [6] i [1 2 3 4 5 6] sum [0 1 3 6 10 15 21] prod [1 1 2 6 24 120 720]

(I1) i=1, (1<=6) T
sum = sum + i = 0+1 = 1
prod = prod * i = 1*1 = 1
i++

(I2) i=2, (2<=6) T
sum = 1+2 = 3
prod = 1*2 = 2
i++

(I3) i=3, (3<=6) T
sum = 3+3 = 6
prod = 2*3 = 6
i++

(I4) i=4, (4<=6) T
sum = 6+4 = 10
prod = 6*4 = 24
i++

(I5) i=5, (5<=6) T
sum = 10+5 = 15
prod = 24*5 = 120
i++

(I6) i=6, (6<=6) T
sum = 15+6 = 21
prod = 120*6 = 720

(I7) i=7, (7<=6) False

Sum(1+2+3+4+5+6) Prod(1*2*3*4*5*6)

13/1/23

⇒ Write a pgm to find Sum of digits of a number.

- expected o/p: i) Enter the number: 5732
  Sum of digits of 5732 = 17
  ii) Enter the number: 10042
  Sum of digits of 10042 = 14

```
import java.util.Scanner;
class SumOfDigits{
P.S.V.m(){
Scanner sc = new Scanner(System.in);
int n, r, sum=0, m;
S.o.pln("Enter the number: ");
n=sc.nextInt();
m=n;
while(n!=0){
r = n%10;
sum = r+sum;
n = n/10;
}
S.o.pln("Sum of digits of "+m+" = "+sum);
}
}
```

Here, we initialize the value of n to m to protect that value (5732) as after looping n become 0.

Initialization is already done, so no need to do it again.

```
for( ; n!=0 ; n=n/10)
{
r = n%10;
sum = r+sum;
}
```

- Tracing:

n	r	sum	m
5732 5732 573 57 5 0	2 3 7 5	0 2 5 12 17	5732

10) 5732 (573 — quotient (/)
32
30
32
8 — remainder (%)

(I1) r = n%10 = 5732%10 = 2
sum = r+sum = 2+0 = 2
n = n/10 = 5732/10 = 573

(I2) n = 573
r = 573%10 = 3
sum = 3+2 = 5
n = 573/10 = 57

(I3) n = 57
r = 57%10 = 7
sum = 7+5 = 12
n = 57/10 = 5

(I4) n = 5,
r = 5%10 = 5
sum = 5+12 = 17
n = 5/10 = 0 (stop)

Pattern programming:

i → rows, j = columns

4:

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

```
for (j=1 ; j<=5 ; j++)
{
  S.o.p("*");
}
```

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

```
for(i=1; i<=5; i++)
{
  S.o.pln("*");
}
```

Nested loops :

When a loop is placed within another loop, the concept is called as Nested loops. Here we go for the next iteration of outer loop only after finishing only after finishing all the iterations of inner loop.

eg: i)

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

outer loop

```
for(i=1 ; i<=5 ; i++)
{ inner loop
  for(j=1 ; j<=5 ; j++){
    S.o.p("*  ");
  }
  S.o.pln ();
}
```

Tabulation of i & j:

i=1 ↓

j=1 j=2 j=3 j=4 j=5 j=6 ×

i=2 ↓

j=1 j=2 j=3 j=4 j=5 j=6 ×

. . . .

.. i=5 ↓

j=1 j=2 j=3 j=4 j=5 j=6 ×

i=6 ×

iii)

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

```
for(i=1 ; i<=5 ; i++){
    for(j=1 ; j<=5 ; j++){
        S.o.p(i +" ");
    }
    S.o.pLn();
}
```

iii)

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

```
for( i=1 ; i<=5 ; i++){
    for (j=1 ; j<=5 ; j++){
        S.o.p(j +" ");
    }
    S.o.pln();
}
```

14|7|23

i=1, j=2 ; 1+2=3

iv)

i=1	2 j=1	3 2	4 3	5 4	6 j=5
i=2	2 j=1	4 2	5 3	6 4	7 5
i=3	4 j=1	5 2	6 3	7 4	8 5
i=4	5 j=1	6 2	7 3	8 4	9 5
i=5	6	7	8	9	10
i=6	7	8	9	10	11

i=6, j=4, 6+4=10

```
for (i=1 ; i<=6 ; i++)
{
    for(j=1 ; j<=5 ; j++)
    {
        S.o.p (i+j+ " ");
    }
    S.o.pln();
}
```

v)

i=1 * j=1 → i=1 (start → end (i=j=1))

i=2 * * j=2 → i=2

i=3 * * * j=3 → i

i=4 * * * * j=4 → i=4

i=5 * * * * * j=5

```
for(i=1 ; i<=5 ; i++)
{
    for(j=1 ; j<=i ; j++)
    {
        S.o.p("* ");
    }
    S.o.pln();
}
```

Tracing: (outer loop iteration → inner loop iteration)

01	(I1)	j<=i	
i=1	j=1 ; 1<=1	—	print("* ");
i<=5	(I2) j=2 ; 2<=1	×	
1<=5			

02

i=2

i<=5

(I1) j=1 ; j<=i ✓ → print("* ");

j=2 ; 2<=2 → print("* ");

j=3 ; 3<=2 ✗

03

i=3

3<=5

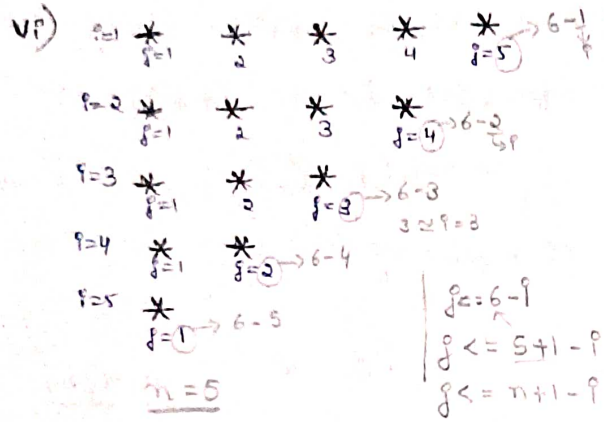
(I1) j=1 ; 1<=3 ✓ → print("* ");

(I2) j=2 ; 2<=3 print("* ");

(I3) j=3 ; 3<=3 print("* ");

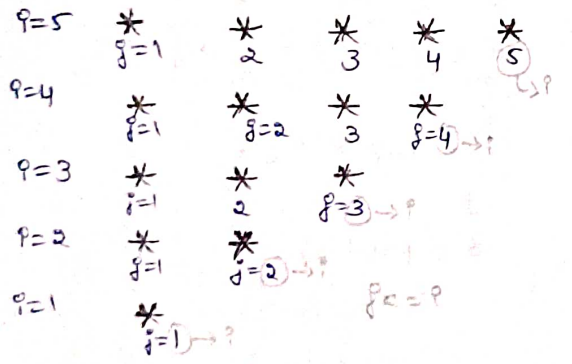
(I4) j=4 ; 4<=3 ✗

vi)



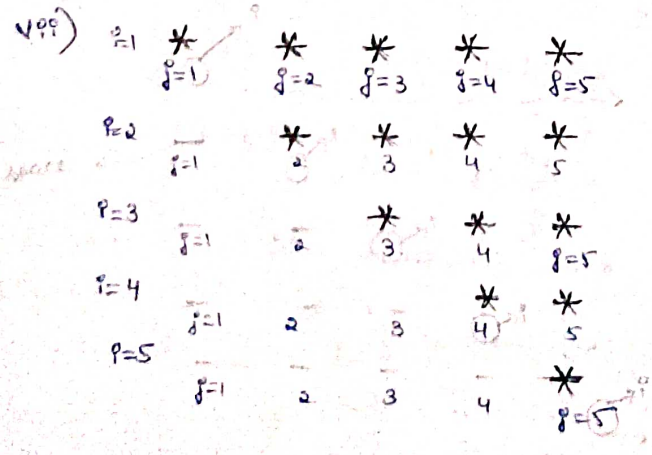

```
for (i=1 ; i<=n ; i++)
{ for (j=1 ; j<=n+1-i ; j++)
  { s.o.p("*  ");
  }
  s.o.pln ( );
}
```

(or)




```
for (i=5 ; i>=1 ; i--)
{
  for (j=1 ; j<=i ; j++)
  {
    s.o.p("*  ");
  }
  s.o.pln();
}
```

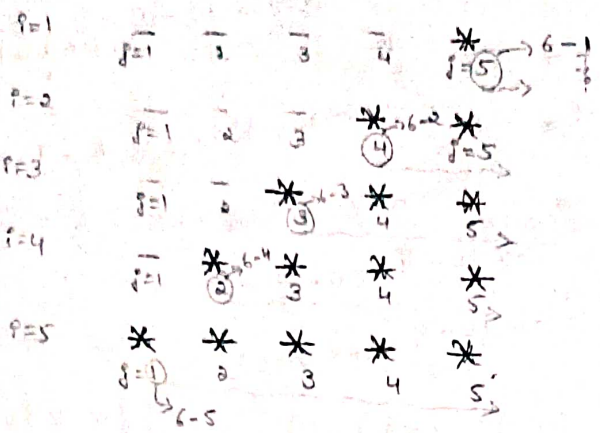
vii)




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

(2 space)

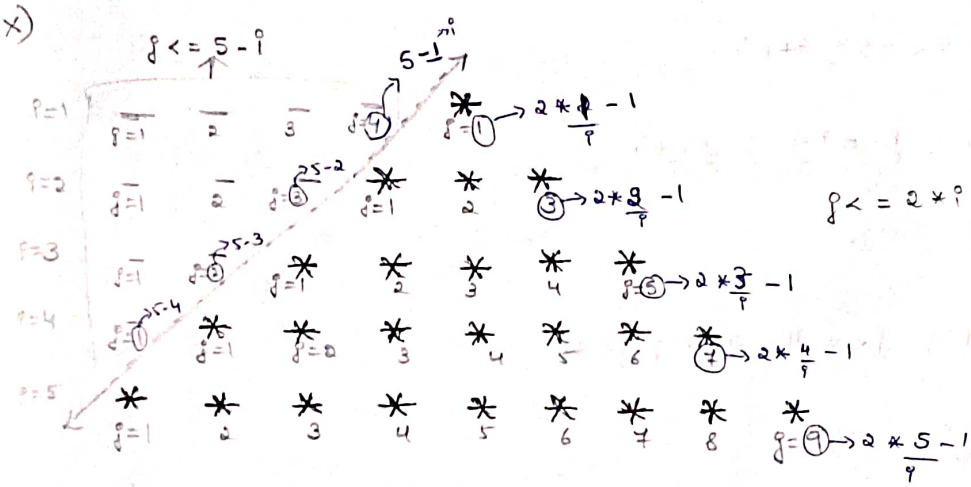
viii)




```
for (i=1; i<=5; i++)
{
 for (j=1; j<=5; j++)
 {
    if (j>=6-i){
     s.o.p ("*  ");
    }
    else{
     s.o.p ("   ");
    }
 }
 s.o.pln();
}
```

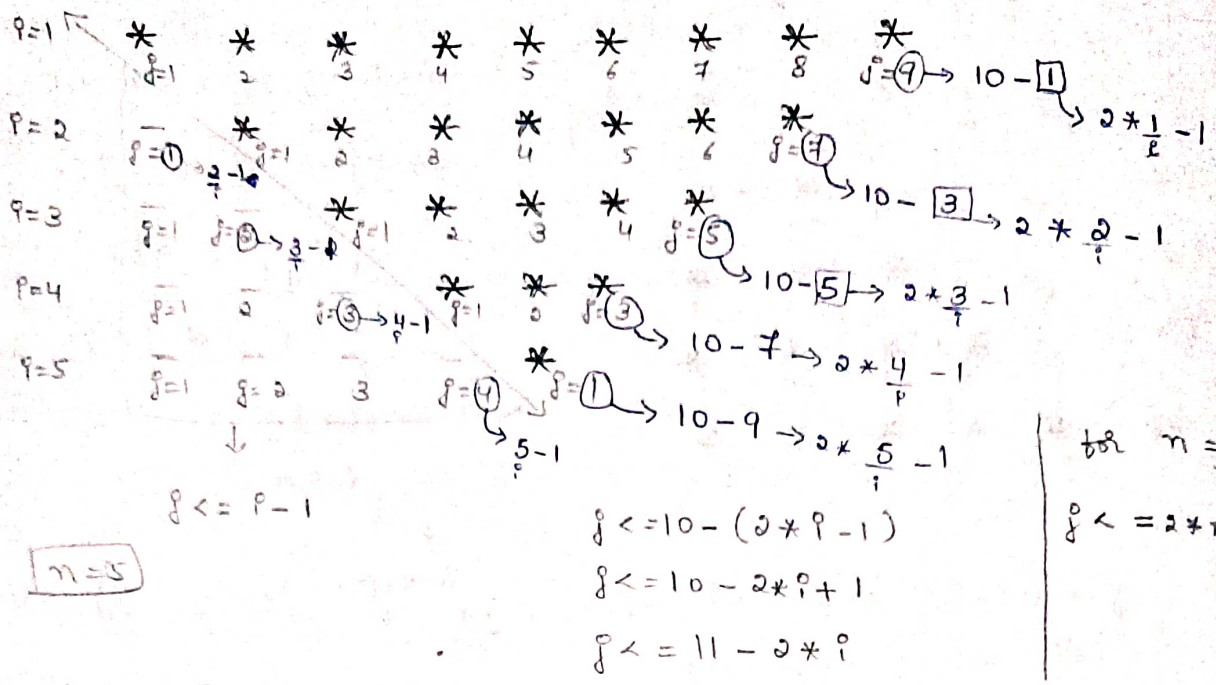
$j >= 6-i$ / $j >= n+1-i$

ix)



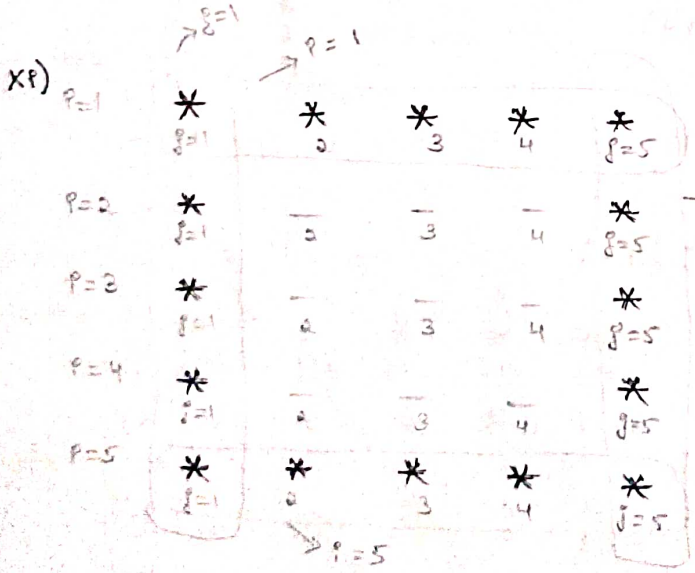

```
for(i=1;  i<=5 ; i++)
{
   for( j=1; j<=5-i ; j++)      (j<=n-i)
   {
     s.o.p("   ");  - space
   }
   for( j=1; j<=2*i-1 ; j++)
   {
     s.o.p("*  ");
   }
   s.o.pln();
}
```

x)



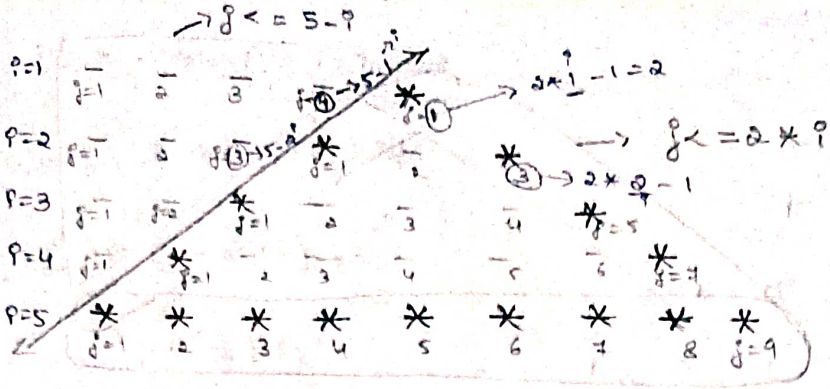

```
for (i=1 ; i<=5 ; i++)
{
  for (j=1 ; j<=i-1 ; j++)
  {
    s.o.p (" ");
  }
  for (j=1 ; j<=11-2*i ; j++)
  {
    s.o.p ("* ");
  }
  s.o.pln ();
}
```

xi)




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

xii)




```
for( i=1 ; i<=5 ; i++){
  for (j=1 ; j<=5-i ; j++)
  {
    s.o.p (" ");
  }
  for(j=1 ; j<=2*i-1 ; j++)
  {
     if ( j==1 || i==5 || j==2*i-1){
        s.o.p("*  ");
     }
     else {
        s.o.p("   ");
     }
  }
  s.o.pln( );
}
```

xiii)

```
i=1   1
      j=1
i=2   2   3
      j=1 2
i=3   4   5   6
      j=1 2   3
i=4   7   8   9   10
      j=1 2   3   4
i=5   11  12  13  14  15
      j=1 2   3   4   5
```

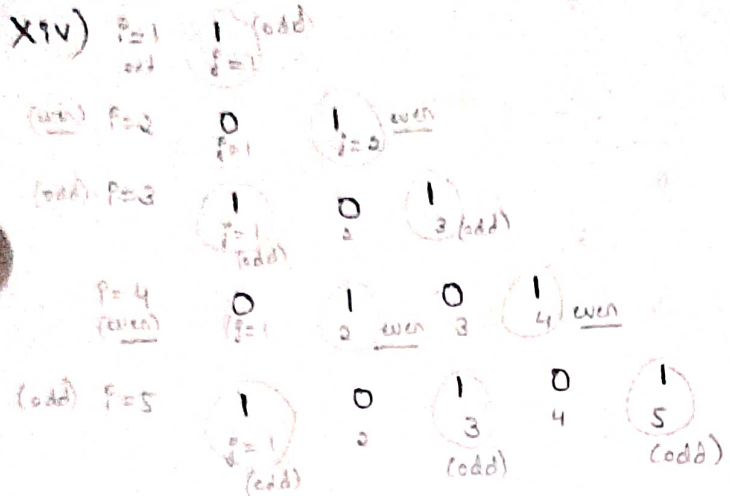
```
count=1 ; ↓
for(i=1 ; i<=5 ; i++)
{                ↓  ↓  ↓  ↓  ↓  9
  for(j=1 ; j<=i ; j++)
  {              ↓  ↓↓  ↓↓↓  ↓↓↓↓  ↓↓↓↓↓  ↓
     s.o.p(count+" ");
     count++;
  }              ↓↓↓↓↓
  s.o.pln( );
}
```

Note :

x ↓ ① → control execution

```
for (i=1 ; i<=5 ; i++){
   ↓  ↓  ↓  ↓  ↓  ↓  ⑤
   for (j=1 ; j<=5 ; j++){
   ↓↓↓↓↓ ↓↓↓↓↓ ↓↓↓↓↓ ↓↓↓↓↓ ↓↓↓↓↓ (25)
   }
   ↓↓↓↓↓ ⑤
}
↓
```

XIV)




```
for(i=1 ; i<=5 ; i++)
{
  for(j=1 ; j<=i ; j++)
  {
    if (i%2==1 && j%2==1)   // odd odd
    {
      S.o.p("1 ");
    }
    else if(i%2==0 && j%2==0)   // even even
    {
      S.o.p("1 ");
    }
    else {
      S.o.p("0 ");
    }
  }
  S.o.pln();
}
```

* when i & j are even (0) → even is represented as zero
  100%2=0, 12%2=0, 4%2=0
  Print 1
* when i & j are odd (1) 99%2=1, 17%2=1
  Print 1
* In other cases print 0.

XV)

a

B c

D e F

g H i J

k L m N o

ASCII (American Standard code for Information Interchange)

```
if(i%2 == j%2)
{
  S.o.p("1 ");
}
else{
  S.o.p("0 ");
}
```

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

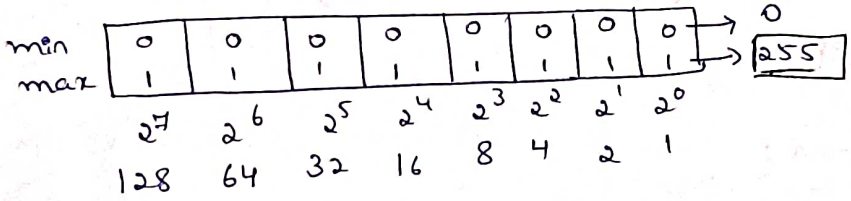
- 5 → Binary → 00000101
- 'a' → 97 → Binary → xxxxxxxx

A → 65	a → 97
B → 66	b → 98
C → 67	.
:	:
Z → 90	z - 122.

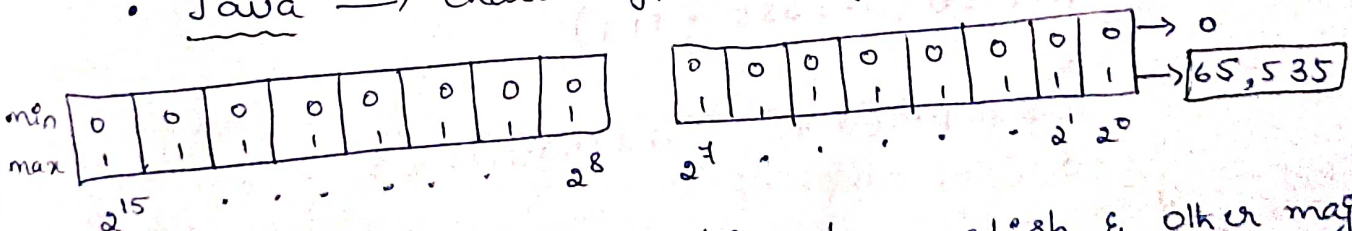
- Even characters must be converted to binary before storing them in memory.
- This is only possible if each character is associated with a no'.
- Hence ASCII values where introduced.

17/7/23

- C → char datatype → 1 byte → Only ASCII



- Java → chardatatype → 2 bytes → ASCII + Unicode



- ASCII had given values for only english & other major European alphabets.
  - → C includes only ASCII values.
  - → Java wanted to include a wide variety of characters. Hence it employed unicode along with ASCII.

xvi)

	64+1	64+2	64+3	64+4	64+5(j)
i=1	A j=1	B 2	C 3	D 4	E j=5
i=2	A j=1	B 2	C 3	D 4	E j=5
i=3	A j=1	B 2	C 3	D 4	E j=5
i=4	A j=1	B 2	C 3	D 4	E j=5
i=5	A j=1	B 2	C 3	D 4	E j=5

```
for(i=1; i<=5; i++)
{ for(j=1; j<=5; j++)
  { s.o.p((char)(64+j)+" ");
  }
  s.o.pln();
}
```

(ASCII for A → 65)

Note: we can obtain ASCII value of a character by type casting it to int & we can obtain characters using ASCII values by type-casting it to 'char'.

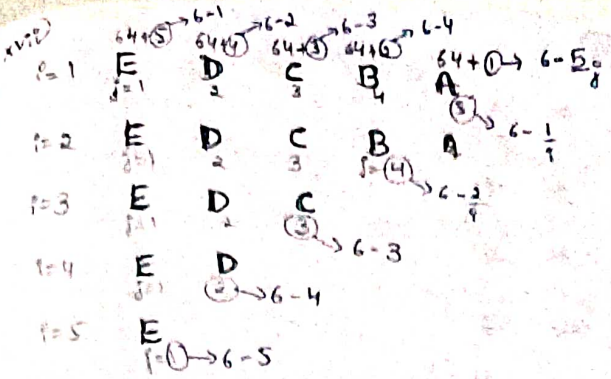
```
int x=65;
s.o.p(x);          // 65
s.o.p((char)x);    // A
```

```
char ch='$'
s.o.p(ch);         // $
s.o.p((int)ch);    // 36
```

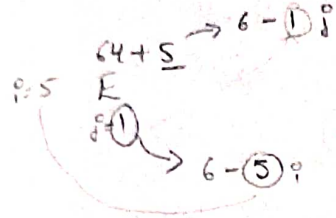
⇒ write a pgm to generate ASCII values & corresponding characters (from 0 → 255).

```
class Ascii
{
  P.S.V.m(){
    int i;
    for(i=0; i<=255; i++)
    {
      s.o.pln(i+"  "+(char)i);
    }
  }
}
```

(xvi)



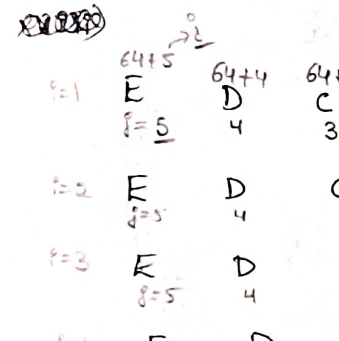

```
i=5
for (i=1; i<=5; i++)
{
   for (j=1; j<=6-i; j++)
   {
      S.o.p ((char) (64+6-j));
   }
   S.o.pln();
}
```




```
i=n
for (i=1; i<=n; i++)
{
   for (j=1; j<=n+1-i; j++)
   {
      S.o.p((char) (64+n+1-j));
   }
   S.o.pln();
}
```

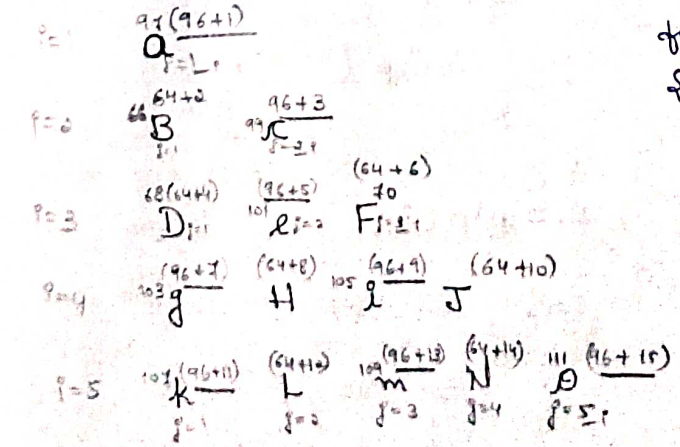
(02)

(xvii)

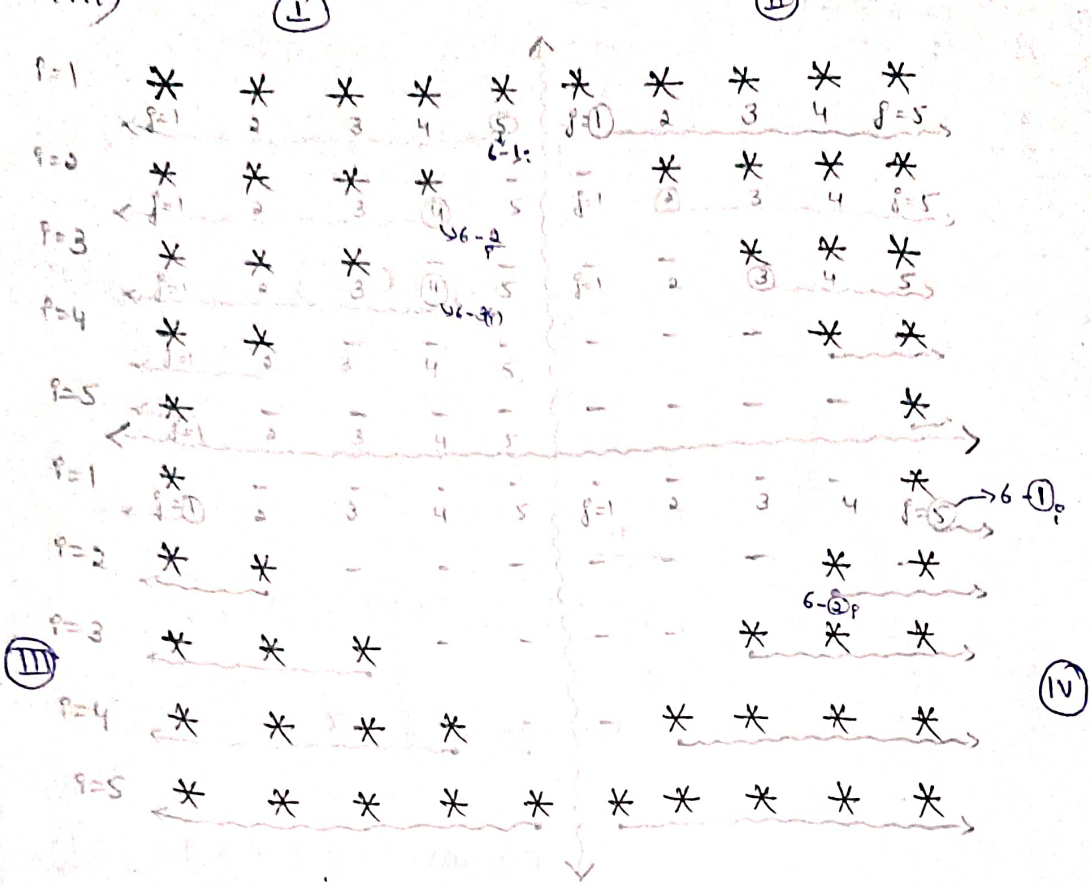



```
for (i=1; i<=5; i++)
{
   for (j=5; j>=i; j--)
   {
      S.o.p((char) (64+j));
   }
   S.o.pln();
}
```

(xv)




```
int count=0, i, j;
for (i=1; i<=5; i++)
{ for (j=1; j<=i; j++)
   { count++;
      if (count%2 == 1){
         S.o.p ((char)(96+count));
      }
      else{
         S.o.p ((char) (64+count));
      }
   }
   S.o.pln();
}
```




```
for(i=1 ; i<=5 ; i++){
  for (j=1 ; j<=5 ; j++){
    if ( j<=6-i ) {
      S.O.P ("*  ");
    }
    else {
      S.O.P ("   ");
    }
  }
  for(j=1 ; j<=5 ; j++){
    if( j >= i ){
      S.O.P ("*  ");
    }
    else {
      S.O.P ("   ");
    }
  }
  S.O.Pln();
}
```

```
for(i=1 ; i<=5 ; i++){
  for (j=1 ; j<=5 ; j++){
    if ( j<=i ) {
      S.O.P("*  ");
    }
    else {
      S.O.P ("   ");
    }
  }
  for (j=1 ; j<=5 ; j++){
    if (j >= 6-i){
      S.O.P ("*  ");
    }
    else {
      S.O.P ("   ");
    }
  }
  S.O.Pln();
}
```

Arrays in Java:

- An Array is a collection of elements of same datatype stored in continuous memory locations.

Disadvantages of using a variable (Limitations of variable):

i) Variables can be used only when the amount of data is very small. As the data increases, no of variables needed will also increase. This makes it difficult to keep track of the variables.

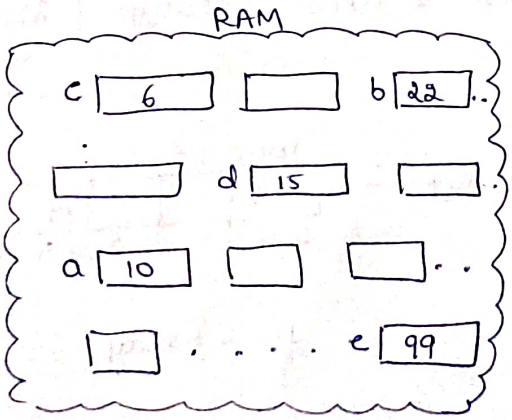
ii) A variable can store only one piece of data at any given point.

eg: int a = 10;
a = 20;
a = 30;
a = 40;
S.o.println(a); //40

a [10 ~~20~~ ~~30~~ 40]

iii) Searching & sorting algorithms require the data to be stored continuously in memory. Hence variables cannot be used to implement these algorithms.

eg: int a=10, b=22, c=6, d=15, e=99;



Types of Arrays:

i) Rectangular Array
   a) 1D Rectangular array (1 row, multiple columns)
   b) 2D R array → multiple rows & columns, each row should have same no. of columns
   c) 3D R array (multiple blocks, rows, columns)

ii) Jagged Array
   a) 2D J-array
   b) 3D J-array

- If the no of columns in each row of an array are equal then we call it as Rectangular array.
- If the no of columns in each row of an array are not equal then it is called Jagged array.

Steps involved in creating an Array:

i) create an array variable.

ii) Using new allocate memory as per requirement.

iii) Address returned by new must be collected by the array variable.

=> create a DataStructure (or Array) to store the marks of 5 students.

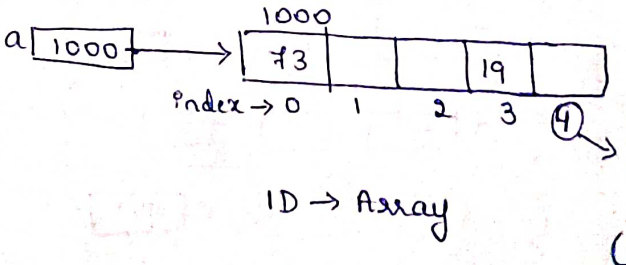
- For the above purpose a 1D-Array with 5 columns should be created as follows:

```
int a[];
a=new int[5];
```
(a ← 1000)

Syntax for creation of 1D Array:

```
Datatype ArrayVariableName[] = new Datatype [ size ]
                                              ↓
                                          (columns)
```

```
a[0]=73;
a[3]=19;
S.o.pln(a[0]);
S.o.pln(a[3]);
```



1D → Array

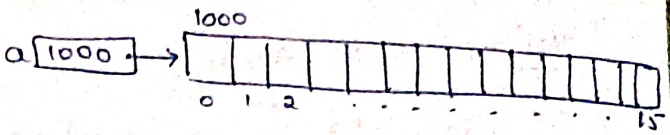
(index = size-1)

Note:
- 1D-array contains only row.
- 2D-array contains multiple rows but only one block.
- 3D-array contains multiple blocks.
- To indicate the position of an element inside an array we use integer values called as index.
- Index always starts from '0'.

- Negative index is not allowed in Java.

⇒ Create an Array to store the marks of 5 students each from 3 classrooms.

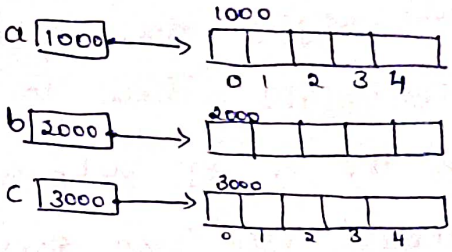
1st soln :



- int a[] = new int[15];
- By using 1st soln it is difficult to keep track of the classrooms.

2nd soln :

- int a[] = new int[5];
  int b[] = new int[5];
  int c[] = new int[5];



- This soln is better than the first soln, As the class rooms can be identified easily but As the no of class rooms increase, the no of arrays needed also increases.

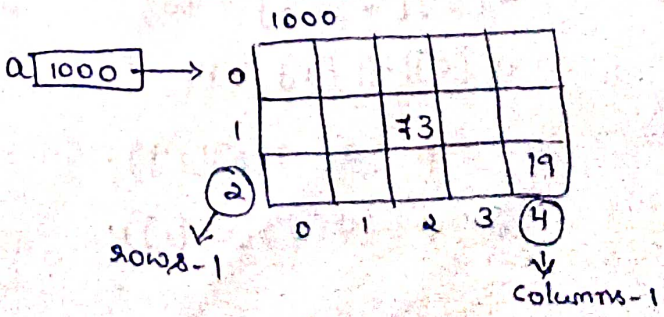
3rd soln : Using 2D-R-array

- int a[][];
  a = new int[3][5];
  (1000)

Syntax for creating 2D Array:

Datatype ArrayVariableName[][] = new Datatype [rows][columns];

```
a[1][2] = 73;
a[2][4] = 19;
S.o.pln (a[1][2]);
S.o.pln (a[2][4]);
```



18/7/23

=> Create an array to store the marks of 5 students each in a classroom & 3 class rooms within each school. The total no of schools are 3.

5(studs) x 3 (class) x 3(schools) = 45

Soln 1: (1D-Array) ✗

```
int a[] = new int[45];
```

→ As the total no of students is 45, we can create 1D Array as shown above. But above soln is not efficient as it is difficult to track the school & class.

Soln 2: (2D-Array) ✗

- ```
  int a[][] = new int[3][5];
  int b[][] = new int[3][5];
  int c[][] = new int[3][5];
  ```
- As the no of schools increase the no of 2D-arrays needed will also increase. Hence the above soln is inefficient.

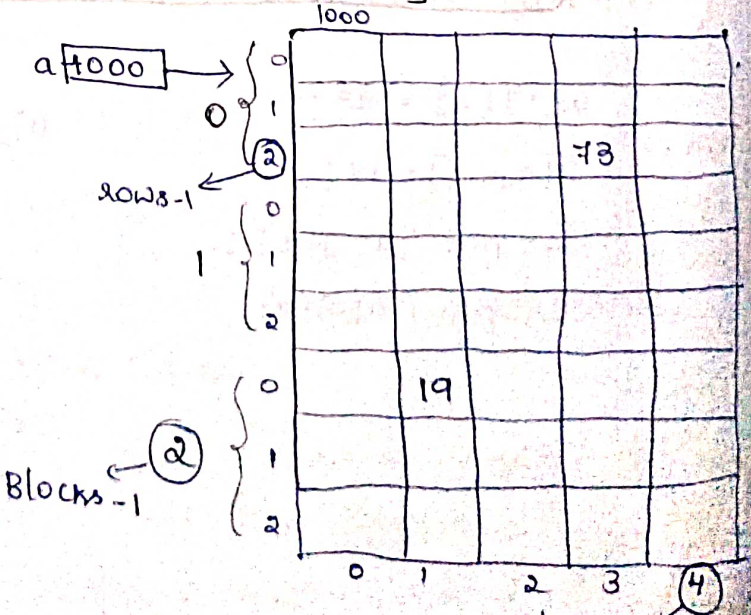
Soln 3: (3D-Array):

- ```
  int a[][][];
  a = new int[3][3][5];
  ```
  (3 → School, 3 → classrooms, 5 → students)

Syntax for creating a 3D-Array:

```
Datatype ArrayVariableName[][][] =
new Datatype [Blocks][Rows][columns]
```

```
a[0][2][3] = 73;
a[0][0][1] = 19;
S.O.p (a[0][2][3]);
S.O.p (a[0][0][1]);
```



⇒ what do you mean by Dimension?

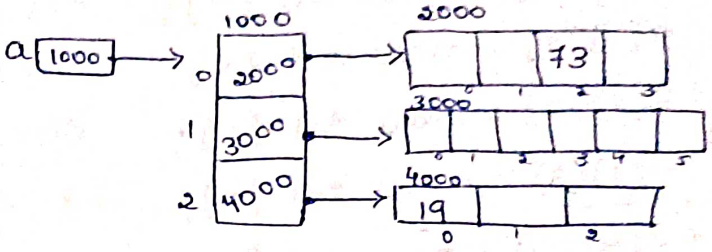
↳ It is the amount of extra information that needs to be provided along with the array variable in order to access an element of the array.

a[ ][ ] → extra info
↓
array variable

⇒ create an array to store marks of students from 3 classrooms. No of students is given below:

class 1 → 4
class 2 → 6
class 3 → 3

```
int a[][];
a = new int[3][ ];   // size of students (columns)
a[0] = new int[4];
a[1] = new int[6];
a[2] = new int[3];
```




```
a[0][2] = 73;
a[2][0] = 19;
S.O.P(a[0][2]);
S.O.P(a[2][0]);
```

• The above array is called as

2D - Jagged Array.

⇒ create an array to store the marks of students from 3 schools. No of class & no of students in each classrooms are as follows:

S1 → C1 (2) C2 (3) C3 (5)

S2 → C1 (5) C2 (4) C3 (2) C4 (4)

S3 → C1 (6) C2 (1)

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

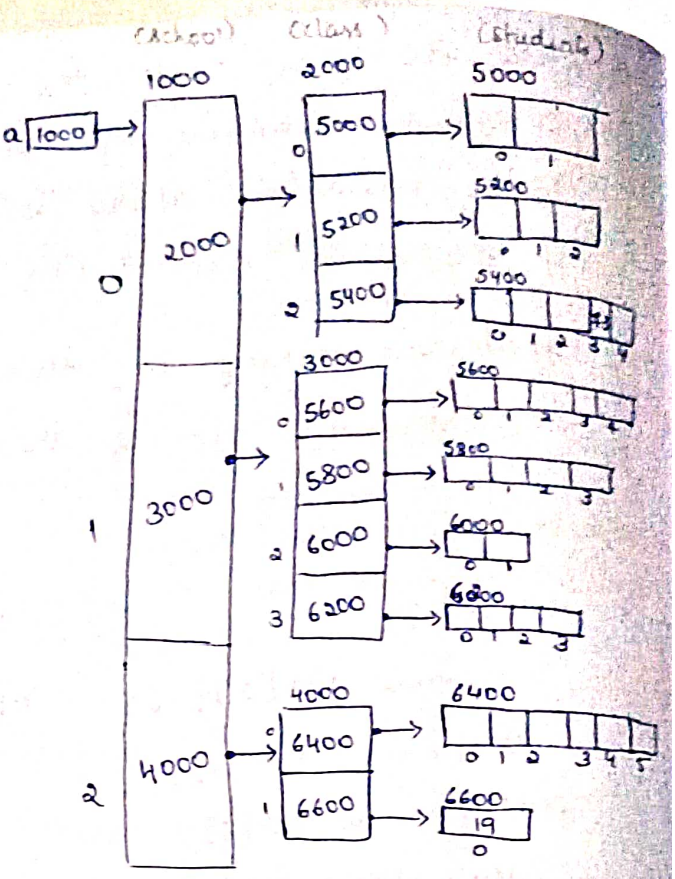
3D - Jagged Array

School 1 → C1 2, C2 3, C3 5 // 3 classrooms

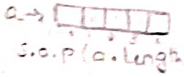
S2 → C1 5, C2 4, C3 2, C4 4 // 4 classrooms

S3 → C1 6, C2 1 // 2 classrooms

```
int a[][][];
a = new int [3][][];
a[0] = new int [3][];
a[1] = new int [4][];
a[2] = new int [2][];
a[0][0] = new int [2];
a[0][1] = new int [3];
a[0][2] = new int [5];
a[1][0] = new int [5];
a[1][1] = new int [4];
a[1][2] = new int [2];
a[1][3] = new int [4];
a[2][0] = new int [6];
a[2][1] = new int [1];
```




```
a[0][2][3] = 73;
a[2][1][0] = 19;
s.o.p (a[0][2][3]);
s.o.p (a[2][1][0]);
```



s.o.p(a.length); → 5

Note:
- The Length variable will give as the size of the array, when the array is of 1D (no of columns).
- When length variable is used with 2D & 3D arrays the output depends on where we use it.

2D Array:

a[i][j] (i: rows, j: columns)

a.length → No of rows

a[i].length → No of columns in row with index = i

1D Array: a[i] (i: column/size)

a.length → No of column size of array.

3D Array:

a[k][i][j] (k: blocks, i: rows, j: columns)

a.length → No of blocks

a[k].length → No of rows in block with index = k

a[k][i].length → No of columns in row with index = i & block with index = k

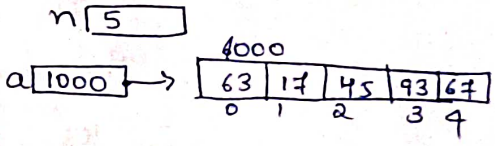
20/7/23

=> create, populate & display 1D Array.

- Expected o/p:

Enter the size of the array : 5 ↵
Enter the marks:
63
17
45
93
67
Marks of Student 1 = 63
Marks of Student 2 = 17
Marks of Student 3 = 45
Marks of Student 4 = 93
Marks of Satudent 5 = 67

- Pgm:




```
import java.util.Scanner;
class OneDArray{
  P.S.V.m(){
    Scanner sc = new Scanner(System.in);
    int n, i;
    S.o.pln ("Enter the size of the array : ");
      n = sc.nextInt();
    int a[] = new int[n];
    S.o.pln ("Enter the marks : ");
    for(i=0; i<=n-1; i++)
                (or) i<a.length-1 (or) i<n
    {
      a[i] = sc.nextInt();
    }
    for(i=0; i<=n-1; i++)
    {
      S.o.pln ("Marks of Student " + (i+1) + " = " +
                                        a[i]);
    }
  }
}
```

If n=5;
a[0] = sc.nextInt();
a[1] = sc.nextInt();
a[2] = sc.nextInt();
a[3] = sc.nextInt();
a[4] = sc.nextInt();

display

⇒ create, populate & display 2D-R-Array.

• Expected o/p:

```
Enter the no of rows : 3 ↵
Enter the no. of columns : 4 ↵
Enter the marks of Students of class 1 :
43
77
82
19
Enter the marks of Students of class 2 :
92
67
40
35
Enter the marks of students of class 3 :
17
47
55
91
The marks of Students are :
43   77   82   19
92   67   40   35
17   47   55   91
```

• Pgm:

```
import java.util.Scanner;
class TwoDArray{
  P.S.V.m(){
    Scanner sc = new Scanner(System.in);
    int rows, cols, i, j;
    S.o.pln("Enter the no. of rows: ");
    rows = sc.nextInt();
    S.o.pln("Enter the no. of columns: ");
    cols = sc.nextInt();
    int a[][] = new int[rows][cols];
    for(i=0; i<=rows-1 ; i++)
    {
      S.o.pln("Enter marks of Students of class "+(i+1)+":");
      for(j=0; j<=cols-1 ; j++)
      {
```

```
            a[i][j] = sc.nextInt();
        }
    }
    S.o.pln("Marks of Students are: ");
    for(i=0; i<=rows-1; i++){
        for(j=0; j<=cols-1; j++){
            S.o.p(a[i][j] + "  ");
        }
        S.o.pln();
    }
}
}
```

- Tracing: memory mapping:

rows | 3 |

cols | 4 |

a | 1000 | → 1000

	j=0	j=1	j=2	j=3
i=0	43	77	82	19
i=1	92	67	40	35
i=2	17	47	55	91

Note:
rows - 1 = a.length - 1
cols - 1 = a[i].length - 1

⇒ create, populate & display 3D-R-Array.

- Expected o/p:

Enter no. of schools : 3 ↵
Enter no. of classes in school 1 : 4 ↵
Enter no. of classes in school 2 : 3 ↵

Enter no. of blocks : 3
Enter no. of rows : 4
Enter no. of columns : 4
Enter marks of Students of School 1 class 1 :
17
32
46
77
Enter marks of Students of school 1 class 2 :
35
46
37
91

school 1 class 3.
school 1 class 4

school 2 class 1
school 3 class 2
class 3
class 4

school 3 class 1
class 2
class 3

:

Enter the marks of students of school 3 class 4:

71
82
63
44

Marks of the students are:

17	32	46	77
35	46	37	91
*	*	*	*
*	*	*	*

- - - - - - - -

*	*	*	*
*	*	*	*
*	*	*	*
*	*	*	*

- - - - - - - - - - - - - -

*	*	*	*
*	*	*	*
*	*	*	*
71	82	63	44

- Pgm:

```
import java.util.Scanner;
class ThreeDArray {
    P.S.V.M() {
        Scanner sc = new Scanner(System.in);
        int rows, cols, blocks, i, j, k;
        S.O.pln ("Enter no. of blocks: ");
        blocks = sc.nextInt();
        S.O.pln ("Enter no. of rows: ");
        rows = sc.nextInt();
        S.O.pln ("Enter no. of columns: ");
        cols = sc.nextInt();
        int a[][][] = new int[blocks][rows][cols];
```

```
for(k=0; k<=blocks-1; k++)
{
  for(i=0; i<=rows-1; i++)
  {
    S.o.pln("Enter marks of Students of school"+
                    (k+1)+"class"+(i+1)+":");
    for(j=0; j<=cols-1; j++)
    {
      a[k][i][j] = sc.nextInt();
    }
  }
}
S.o.pln("Marks of students are:");
for(k=0; k<=blocks-1; k++){
  for(i=0; i<=rows-1; i++)
  {
    for(j=0; j<=cols-1; j++)
    {
      S.o.p(a[k][i][j]+" ");
    }
    S.o.pln();
  }
  S.o.pln("----------");
}
}
```

- mapping: blocks 3 rows 4 columns 4

a 1000 →

K=0	i=0	17	32	46	71
	1	35	46	37	91
	2				
	3				
K=1	0				
	1				
	2				
	3				
K=2	0				
	1				
	2				
	3	71	82	63	44
		j=0	1	2	3

=> Create, populate & display 2D - Jagged Array.

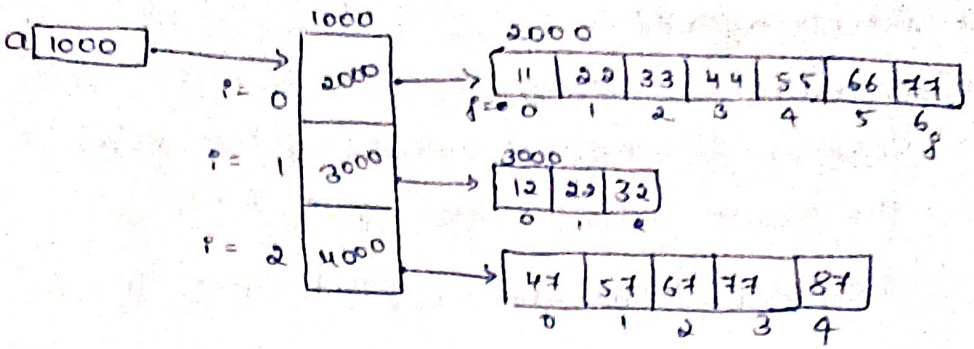
c1 → 7 || 22 33 44 55 66 77

c2 → 3 || 12 22 32

c3 → 5 || 41 51 61 71 81

```
import java.util.Scanner;
class TwoDJagged{
  P.S.V.m(){
    Scanner sc=new Scanner(System.in);
    int rows, cols, i, j;
    S.o.pln("Enter no. of classes: ");
    rows = sc.nextInt();
    int a[][] =new int[rows][ ];
    for(i=0; i<=rows-1; i++){
      S.o.pln("Enter no. of students in class"+(i+1)+": ")
      cols = sc.nextInt();
      a[i] = new int[cols];
    }
    for(i=0; i<= a.length - 1 ; i++){
      S.o.pln("Enter the marks of students of class "+(i+1)+" ");
      for(j=0; j<= a[i].length - 1; j++){
        a[i][j] = sc.nextInt();
      }
    }
    S.o.pln("Marks of the students are: ");
    for(i= 0; i<= a.length-1 ; i++) {
      for(j=0; j<= a[i].length-1 ; i++){
        s.o.p (a[i][j] + " ");
      }
      S.o.pln();
    }
  }
```

- Mapping: rows [5] cols [7 8 5]



- Output :

Enter the no. of classes : 3
Enter the no. of students in class1 : 7
Enter the no. of students in class2 : 3
Enter the no. of students in class3 : 5
Enter the marks of student of class 1 :
11 22 33 44 55 66 77
Enter the marks of students of class2 :
12 22 32
Enter the marks of students of class3:
47 57 67 77 87

Marks of the students are :
11 22 33 44 55 66 77
12 22 32
47 57 67 77 87

⇒ create, populate & display 3D - Jagged Array.

S1 C1 → (7) 11 21 31 41 51 61 71
C2 → (3) 12 22 32
C3 → (5) 13 23 33 43 53

S2 C1 → (4) 14 24 34 44
C2 → (3) 15 25 35

S3 C1 → (6) 16 26 36 46 56 66
C2 → (2) 12 22
C3 → (4) 18 28 38 48
C4 → (7) 19 29 39 49 59 69 79

```
import java.util.Scanner;
class ThreeDJagged{
    P.S.V.m(){
        Scanner sc = new Scanner(System.in);
        int rows, cols, blocks, j, i, k;
        S.o.pln (" Enter no. of schools : ");
        blocks = sc.nextInt();
        int a[][][] = new int[blocks][][];
        for(i=0 ; i<=blocks-1 ; i++){
            S.o.pln ("Enter no. of classes in school "+(i+1)+":
            rows = sc.nextInt();
            a[i] = new int[rows][];
        }
        for(i=0 ; i<=a.length-1 ; i++){
            for(j=0; j<=a[i].length-1 ; j++){
                S.o.pln ("Enter no. of students in school "+(i+1)+
                                "class "+(j+1)+" :");
                cols = sc.nextInt();
                a[i][j] = new int[rows][cols];
            }
        }
        for(k=0 ; k<=a.length-1 ; k++){
            for(i=0 ; i<=a[k].length-1 ; i++){
                S.o.pln("Enter marks of students of school "+(k+
                                "class "+(i+1)+" :");
                for(j=0; j<=a[k][i].length-1 ;j++){
                    a[k][i][j] = sc.nextInt();
                }
            }
        }
    }
```

```
S.o.pln("Marks of the students are : ");
for(k=0; k<=a.length-1; k++){
    for(i=0; i<=a[k].length-1; i++){
        for(j=0; j<=a[k][i].length-1; j++){
            S.o.p(a[k][i][j] + "  ");
        }
        S.o.pln();
    }
    S.o.pln("-----------------------");
}
}
}
```

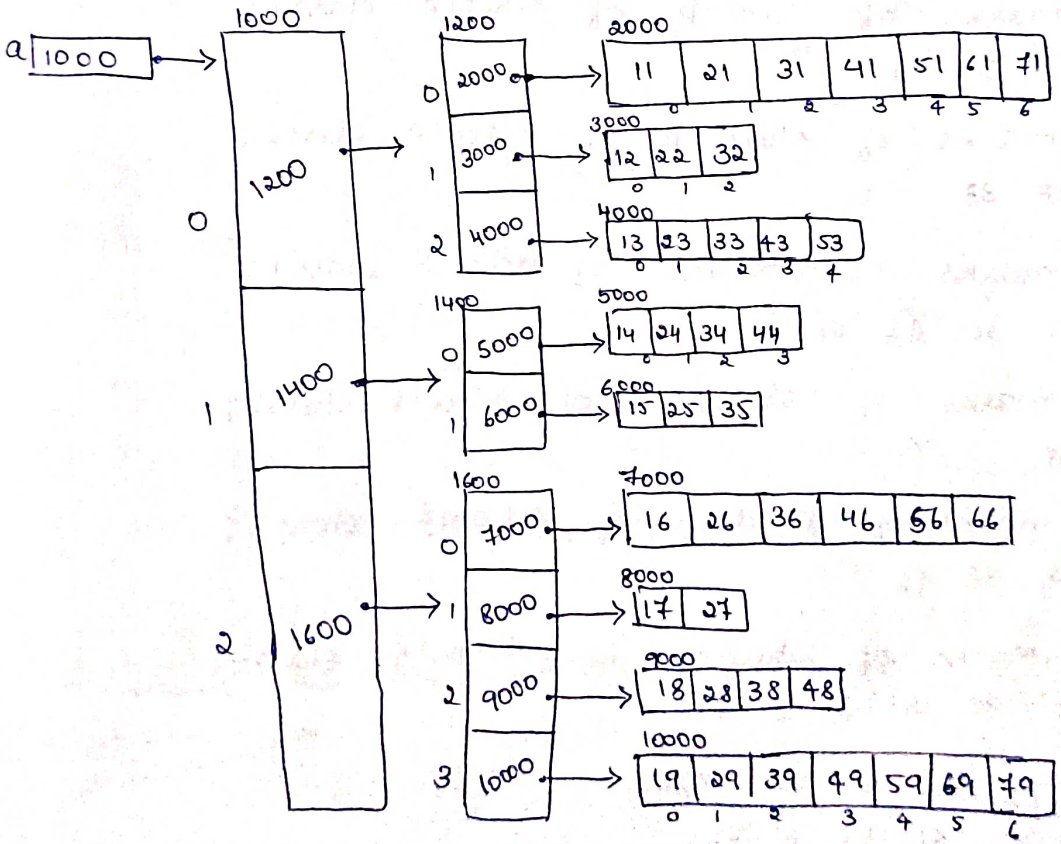
display

- memory Mapping

blocks 3

rows ~~3~~ ~~2~~ 4

cols ~~7~~ ~~3~~ ~~8~~ ~~4~~ ~~3~~ ~~6~~ ~~2~~ ~~4~~ 7



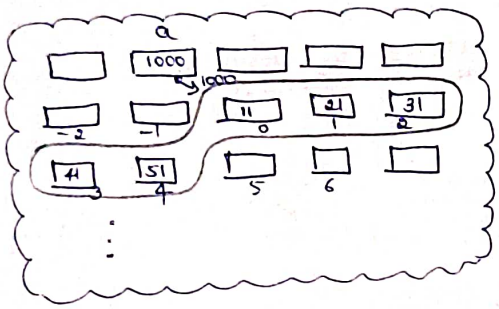
- Output :

```
Enter the no. of schools : 3
Enter the no. of classes in school1 : 3
Enter the no. of classes in school2 : 2
Enter the no. of classes in school3 : 4
Enter no. of students in school1 class1 : 7
Enter no. of students in school1 class2 : 3
```

```
Enter the no. of students in school1 class3 : 5
Enter the no. of students in school2 class1 : 4
Enter the no. of students in school2 class2 : 3
Enter the no. of students in school3 class1 : 6
Enter the no. of students in school3 class2 : 2
Enter the no. of students in school3 class3 : 4
Enter the no. of students in school3 class4 : 7

Enter marks of students of school1 class1 :
 11 21 31 41 51 61 71
Enter marks of students of school1 class2 :
 12 22 32
Enter marks of students of school1 class3 :
 13 23 33 43 53
Enter marks of students of school2 class1 :
 14 24 34 44
Enter marks of students of school2 class2 :
 15 25 35
Enter marks of students of school3 class1 :
 16 26 36 46 56 66
Enter marks of students of school3 class2 :
 17 27
Enter marks of students of school3 class3 :
 18 28 38 48
Enter marks of students of school3 class4 :
 19 29 39 49 59 69 79

Marks of the students are :
 11 21 31 41 51 61 71
 12 22 32
 13 23 33 43 53
 14 24 34 44
 15 25 35
 16 26 36 46 56 66
 17 27
 18 28 38 48
 19 29 39 49 59 69 79
```

Buffer OverRun:

- When we try to access the memory location beyond what is allocated for us, BufferOverRun occurs.
- This is not allowed in Java as array boundaries are strictly enforced.
- If we try to access an array index less than '0' & greater than 'size-1', an ArrayIndexOutOfBoundsException is generated.




```
int a[] = new int[5];
a[0] = 11;
a[1] = 21;
a[2] = 31;
a[3] = 41;
a[4] = 51;
a[5] = 61;
a[6] = 71;
```
} // AIOOBE

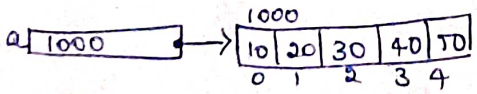
Ways of declaring Array:

(i) `int a[] = new int[5];`
(ii) `int []a = new int [5];`
(iii) `int[] a = new int [5];`
} Declaration

(iv) `int a[] = {1, 2, 3, 4, 5, 6};`
(v) `int a[] = new int[]{1, 2, 3, 4, 5, 6};`
} Declaration + Initialization

24/7/23

Enhanced for loop: (forEach loop):




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

- Initialization
- Condition
- Accessing
- print
- Inc / Dec

→ Syntax:

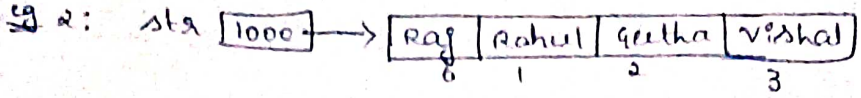
```
for(Datatype var : ArrayVariable)
{
  S1;
  S2;
  :
  Sn;
}
```

(10, 20, 30, 40, 50)

```
eg: for (int x : A) {
      S.O.P (x);
    }
```

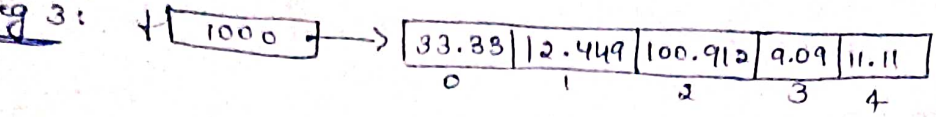
- Initialization
- print

Eg 2:




```
for(String s : str){
    S.o.p(s);
}
```

Eg 3:




```
for (float a : f){
    S.o.p(a);
}
```

=> Write a pgm to create an array of objects.

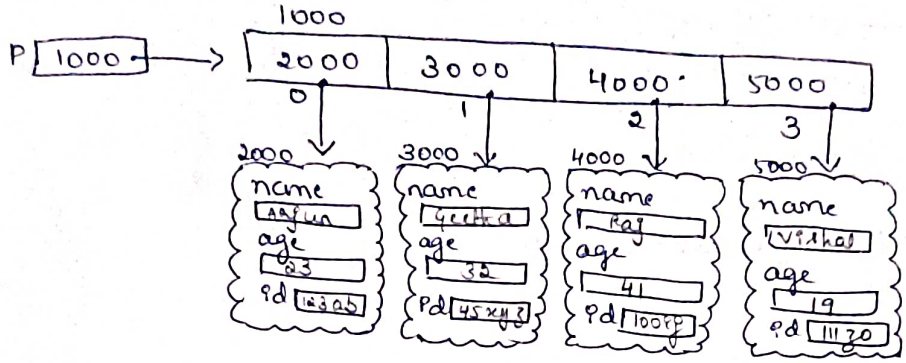
```
class Person{
    String name;
    String id;
    int age;
}
import java.util.Scanner;
class PersonApp{
  P.S.V.m(){
    Scanner sc = new Scanner(System.in);
    S.o.pln ("Enter no. of Person objects to be created:");
    int n = sc.nextInt();
    Person P[] = new Person[n];
    int i;
//---
    for(i=0; i<=n-1; i++)
    {
      P[i] = new Person();
    }
//---
    for(i=0; i<=n-1; i++)
    {
      S.o.pln ("Enter name of person" + (i+1) + ":");
      P[i].name = sc.nextLine();
      S.o.pln ("Enter age of person" + (i+1) + ":");
      P[i].age = sc.nextInt();
      S.o.pln ("Enter id of person" + (i+1) + ":");
      P[i].id = sc.next();
    }
```

creation

population

```
S.o.pln ("Details of the persons are :");
for(i=0; i<=n-1; i++)
{
    S.o.pln (p[i].name);
    S.o.pln (p[i].age);
    S.o.pln (p[i].id);
    S.o.pln ("-------------");
}
}
}
```

display



"Collection"

Default Values of Instance Variables :

- Unlike local variables, instance variables donot contain garbage values by default.
- Based on the datatype of the instance variable corresponding values will be stored within them.

```
import class Student{
        byte b;
        short s;
        int i;
        long l;
        float f;
        double d;
        char c;
        boolean bool;
        String str;
    }
class InstanceDefault{
        P.S.V.m(){
        Student s1 = new Student[];
        S.o.pln (s1.b);
```

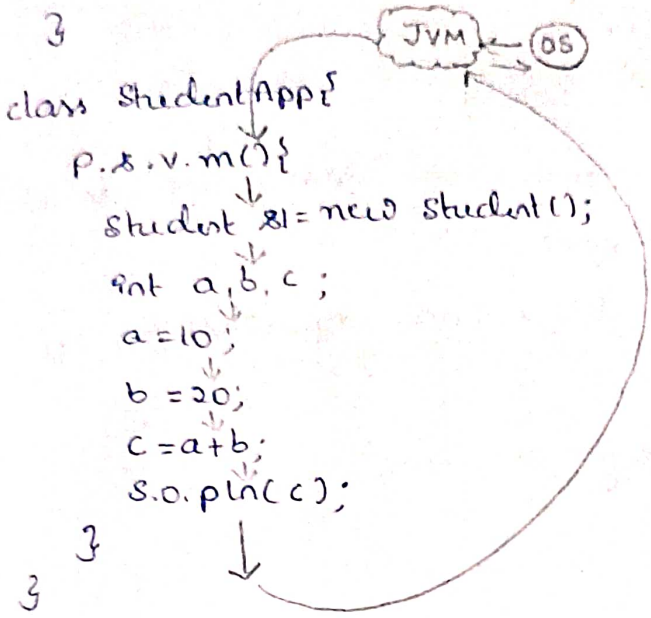
```
        S.O.pln(S1.s);
        S.O.pln(S1.i);
        S.O.pln(S1.l);
        S.O.pln(S1.f);
        S.O.pln(S1.d);
        S.O.pln(S1.c);
        S.O.pln(S1.bool);
        S.O.pln(S1.str);
    }
}
```

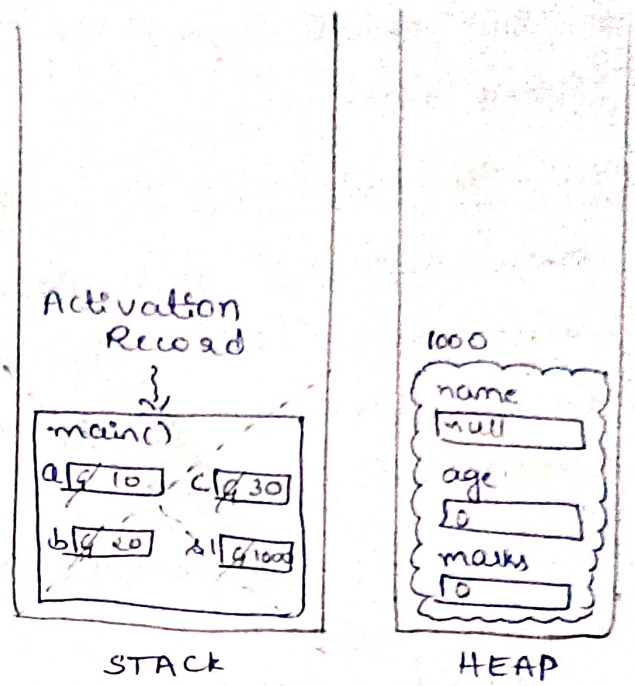
o/p:
```
0
0
0
0
0.0
0.0
'\0' ⟶ null character = '\0'
false
null
```

Instance Variables v/s Local Variables:

Instance Variables	Local Variables
• Declared within class outside the methods.	• Declared within methods.
• Default value is not garbage.	• Default value is garbage.
• Allocated memory on Heap within objects	• Allocated memory on Stack within Activation records.
• Allocated memory during Object creation.	• Allocated memory when control enters the method.
• Deallocated memory when Object becomes garbage.	• Deallocated memory when method execution completes & control exists the method

```
class Student {
    String name;
    int age;
    int marks;
}
class StudentApp {
    p.s.v.m() {
        Student s1 = new Student();
        int a, b, c;
        a = 10;
        b = 20;
        c = a+b;
        S.O.pln(c);
    }
}
```





- When the control enters into the method, it becomes active & an Activation Record is allocated dedicatedly for the method on the stack.
- Activation record is deallocated when method execution completes & control exits the method.

25/7/23

Strings:

- In Java String is a collection of characters placed within double quotations (" ").
- Here we must note that, string is not an array of characters.
- String is a class in Java & hence a non-primitive datatype.

There are 2 types of Strings:

i) Mutable Strings: mutable strings are those whose content can be changed after initialization.

- we can create mutable strings by creating objects of StringBuilder class (or) StringBuffer class.

ii) Immutable strings : Immutable strings are those where values cannot be changed after initialization.

- We can create immutable strings by creating objects of String class

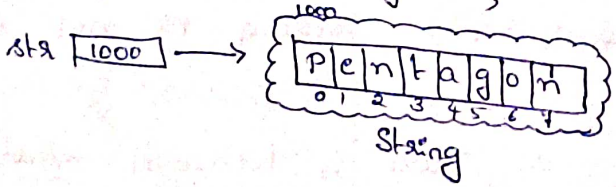
Need for mutable & immutable Strings :

- While designing applications there are some fields where values should never be changed.
- Let us take the eg of a Bank Loan Application to understand the above.

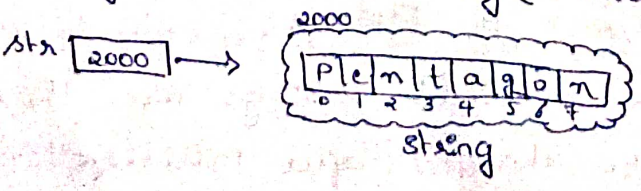
Name → Immutable String
Age → int
Address → Mutable String
DOB → Immutable string
Ph. No → Mutable string
Email → Mutable String
Aadhar → Immutable String
PAN → Immutable String
Salary → int
Designation → Mutable String

2 ways of Creating Immutable String :

Type - 1 : String str = "Pentagon";



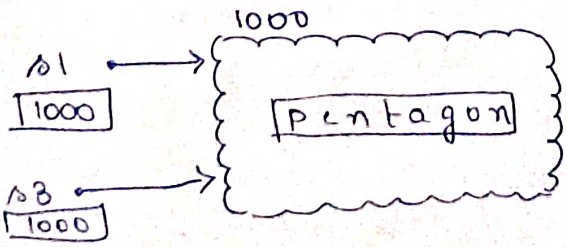
Type - 2 : String str = new String("Pentagon");

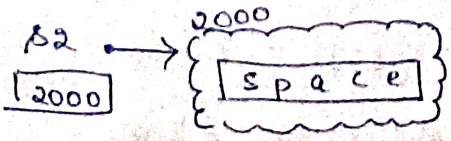


Concept-1 : Comparing content of 2 Strings : (equals())

```
class Demo{
  P.S.V.m(){
    String s1 = "Pentagon";
    String s2 = "Space";
    String s3 = "Pentagon";
    if (s1.equals(s2)){
      S.O.pln("content of s1 & s2 are same");  (F)
    }
    else{
      S.O.pln("content of s1 & s2 are not same");  (T) ✓
    }
    if (s1.equals(s3){
      S.O.pln("content of s1 & s3 are same");  (T) ✓
    }
    else{
      S.O.pln("content of s1 & s3 are not same");
    }
    if (s2.equals(s3){
      S.O.pln("content of s2 & s3 are same");  (F)
    }
    else{
      S.O.pln("content of s2 & s3 are not same");  (T) ✓
    }
  }
}
```

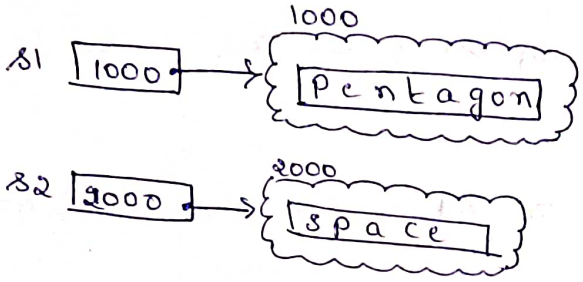
equals() → True [content of 2 strings are equal]
→ False [content of 2 strings are not equal]





Concept2 : comparing the addresses within 2 string address variables (==)

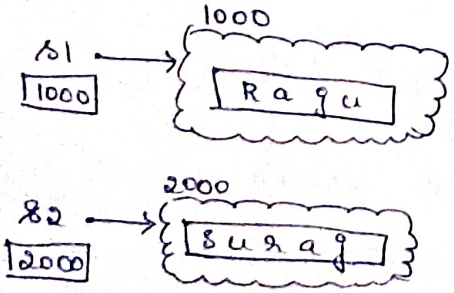
```
class Demo {
  P.S.V.m(){
    String s1 = "Pentagon";
    String s2 = "space";
    if (s1 == s2) {          // (1000) (2000)
      S.o.pln("s1 & s2 are pointing to same string object"); (F)
    }
    else {
      S.o.pln("s1 & s2 are pointing to different object"); T(—)
    }
  }
}
```



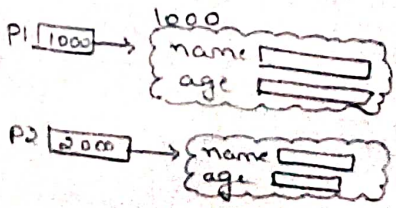
s1, s2 → string address variables (&) string variables

Concept-3 : Printing string address variable.

```
class Demo{
  P.S.V.m(){
    String s1 = "Raju";
    String s2 = "Suraj";
    S.o.pln(s1);    // Raju
    S.o.pln(s2);    // Suraj
  }
}
```




```
class Person{
  String name;
  Int age;
}
class Demo{
  P.S.V.m(){
    Person p1 = new Person();   // 1000
    Person p2 = new Person();   // 2000
    S.o.pln(p1);    // 1000
    S.o.pln(p2);    // 2000
  }
}
```

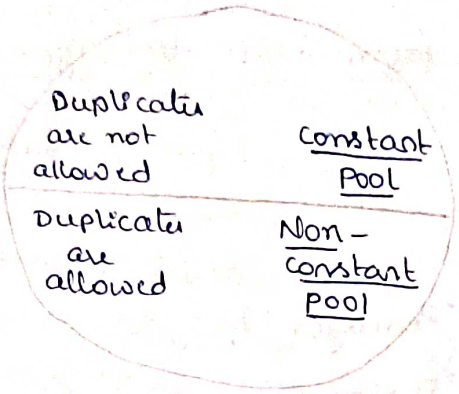


Note: S.o.pln(p1) → S.o.pln(p1.toString()); // 1000

S.o.pln(s1); → S.o.pln(s1.toString()); // Raju

- Whenever we place address variable of a class within S.o.p (or) S.o.pln, toString() method of the class is invoked.
- Depending on the implementation of toString(), the output changes.

Concept-4: Memory for strings comes from String pool on heap.

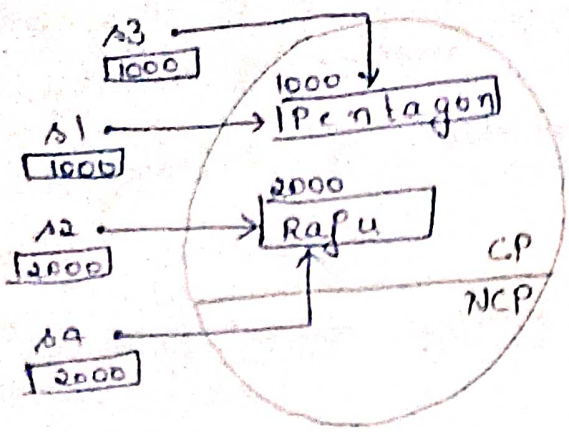


String Pool

As no duplicates are allowed in constan pool, only one copy of an object will be created & multiple address variables are made to point to same copy whenever required.

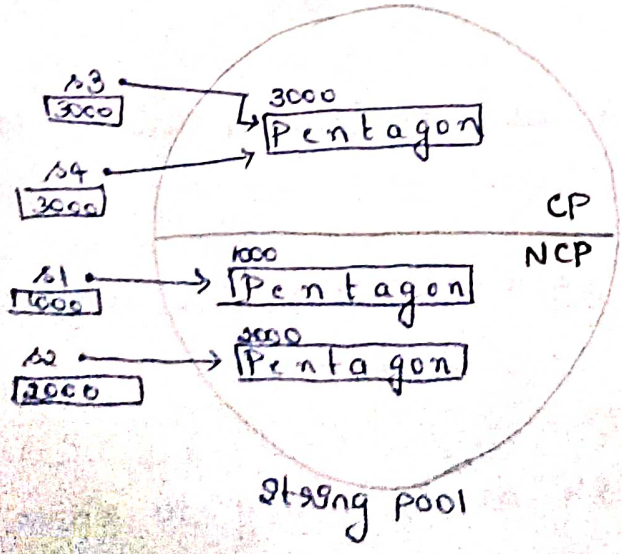
Concept-5: Memory comes from CP for Type-1 declaration:

```
Class Demo{
 P.S.V.m(){
   String s1 = "Pentagon";
   String s2 = "Raju";
   String s3 = "Pentagon";
   String s4 = "Raju";

   if (s1 == s3){
    S.o.pln ("s1 & s3 are pointing to same object");   (T)
   }
   else{
    S.o.pln ("s1 & s3 are pointing to different objects");
   }
   if (s2 == s4){
    S.o.pln("s2 & s4 are pointing to same object");   (T)
   }
   else{
     S.o.pln ("s2 & s4 are pointing to different objects");
   }
 }
}
```



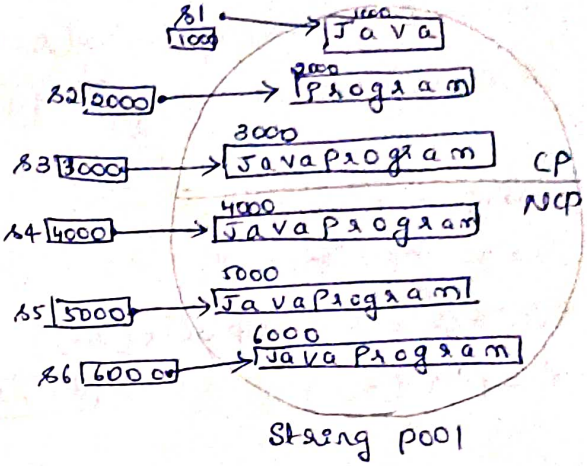
Concept - 6 : Memory comes from NCP for Type - 2 declaration of strings.

```
Class Demo{
  P.S.V.m(){
    String s1=new String("Pentagon");
    String s2=new String("Pentagon");
    String s3="Pentagon";
    String s4="Pentagon";
    if(s1==s2) →(F)
    if(s1==s3) (F)
    if(s1==s4) F
    if(s2==s3) F
    if(s2==s4) F
    if(s3==s4) (T)
  }
}
```



Concept 7: How memory is allocated during concatenation?

```
String s1 = "Java";
String s2 = "Program";
String s3, s4, s5, s6;
s4 = s1 + s3 = "Java" + "Program";
s4 = s1 + "Program";
s5 = "Java" + s2;
s6 = s1 + s2;
S.O.pln(s1); // Java
S.O.pln(s2); // Program
S.O.pln(s3); // JavaProgram
S.O.pln(s4); // JavaProgram
S.O.pln(s5); // JavaProgram
S.O.pln(s6); // JavaProgram
```



Note:

- Values + Values
  Java + Program } → CP

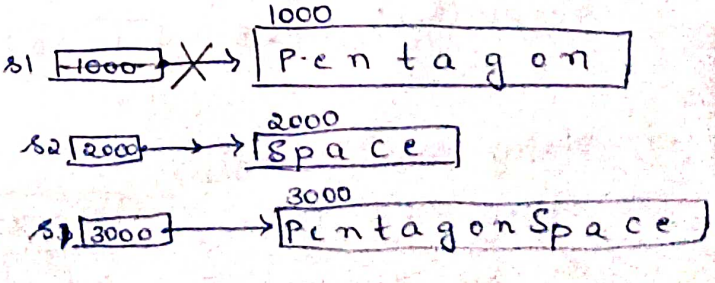
- Value + Variable
  "Java" + s2
- Variable + Value
  s1 + "Program"
- Variable + Variable
  s1 + s2

} → NCP

- If variables is used in concatination, memory comes from NCP.

Concept-8:

```
String s1 = "Pentagon";
String s2 = "Space";
s1 = s1 + s2;
S.O.pln(s1); //PentagonSpace
```



- In the above eg, the rules of Immutability have not been broken. String "Pentagon" has not been changed to "PentagonSpace", rather they are 2 seperate objects.
- s1 which was pointing to address 1000 is now pointing to address 3000.

Three-way Comparision of Strings:

ASCII
A - 65
B - 66
:
Z - 90

compareTo() → [s1 > s2] / [s1 == s2] / [s1 < s2]

=> String s1 = "Ram u";
String s2 = "Rama";
int res = s1.compareTo(s2);
S.o.pln(res); // 20 — s1 > s2

ASCII

R	a	m	u
(82)	(97)	(109)	(117)
R	a	m	a
(82)	(97)	(109)	(97)
0	0	0	+20

=> String s1 = "bala";
String s2 = "Bala";
int res = s1.compareTo(s2);
S.o.pln(res) // 32 — s1 > s2

b	a	l	a
(98)	(97)	(108)	(97)
B	a	l	a
(66)	(97)	(108)	(97)
32			

=> String s1 = "yuva";
String s2 = "yuvi";
int res = s1.compareTo(s2);
S.o.pln(res); // -8 — s2 > s1

y	u	v	a
(89)	(117)	(118)	(97)
y	u	v	i
(89)	(117)	(118)	(105)
0	0	0	-8

=> String s1 = "Bala";
String s2 = "Balaji";
int res = s1.compareTo(s2);
S.o.pln(res); // -5 — s1 < s2

27/7/23

s1 = Bala
s2 = Balaji
(-)
-2 → length diff

=> String s1 = "Balaji";
String s2 = "Bala"; — s1 > s2
int res = s1.compareTo(s2);
S.o.pln(res); // +2

B a l a j i
(-) B a l a
+2

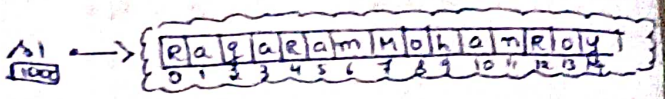
=> String s1 = "Balaji";
String s2 = "bala"; — s1 < s2
int res = s1.compareTo(s2);
S.o.pln(res); // -32

(66) B a l a j i
(98) b a l a
-32

Direct indexing of strings not allowed:

```
String s1 = "Rakesh";
s1[0] = 'h';
S.o.pln(s1[0]); } Error
```

String Methods:




```
class StringMethods {
    P.S.V.m() {
        String s1 = "RajaRamMohanRoy";
        S.O.pln(s1.length());          // 15
        S.o.pln(s1.charAt(12));        // R
        S.o.pln(s1.indexOf('R'));      // 0
        S.o.pln(s1.lastIndexOf("R"));  // RAJARAMMOHANROY 12
        S.o.pln(s1.toUppercase());     // RAJA RAMMOHANROY
        S.o.pln(s1.toLowerCase());     // rajarammohanroy
        S.o.pln(s1.substring(7));      // MohanRoy
        S.o.pln(s1.substring(4, 11));  // RamMoha
    }
}
```

Note:

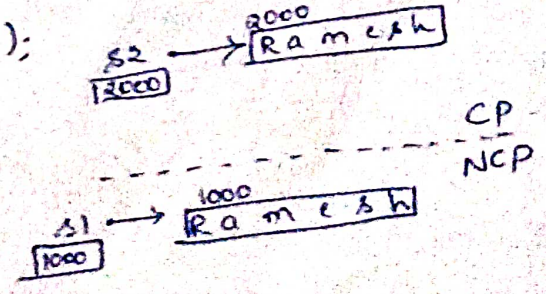
Other String methods:

i) equals()
ii) compareTo()
iii) equalsIgnoreCase() → Used to compare content of 2 strings case insensitively

```
eg: s1 = "Raj";
    s2 = "raj";
    s1.equals(s2);                  // False
    s1.equalsIgnoreCase(s2);        // True
```

(iv) intern() → the process of creating the copy of a String present in NCP onto the CP is called intern().

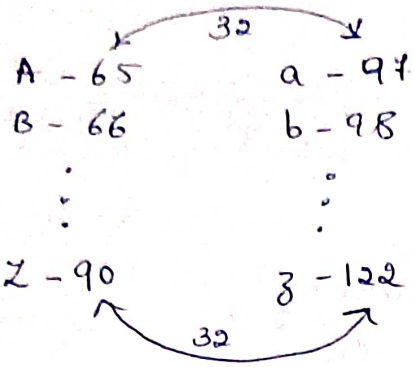
```
eg: String s1 = new String("Ramesh");
    String s2 = s1.intern();
```



=> Write a pgm to swap / switch the case of given string.

E.g:
Enter string : Pentagon Space ;

After swapping case : pENTAGON sPACE




```
import java.util.Scanner;
class SwapCase{
    P.S.V.M(){
        Scanner sc = new Scanner(System.in);
        S.O.pln("Enter string : ");
        String str = sc.nextLine();
        int i, ascii ;
        char ch;
        for(i=0; i<=str.length()-1 ; i++){
         ch = str.charAt(i);
         ascii = (int)ch;
         if(ascii >=65 && ascii <=90){
            ascii = ascii + 32;
         }
         else if(ascii >=97 && ascii <=122){
            ascii = ascii - 32;
         }
         S.O.P ((char) ascii);
    }
}
}
```

Tracing :

str →	H	i	!		J	O	h	n
	0	1	2	3	4	5	6	7

str.length()-1
8-1 = 7

(I1) i=0, 0<=7 (T)
ch = str.charAt(0) → 'H'
ascii = (int)'H' → 72
if (72 >= 65 && 72 <= 90) (T)
ascii = 72+32 = 104
print((char) 104) → h

(I2) i=1, 1<=7 (T)
ch = str.charAt(1) → 'i'
ascii = (int)'i' → 105
if (105 >= 65 && 105 <= 90) (F)
elseif (105 >= 97 && 105 <= 122) (T)
ascii = 105 - 32 = 73
print((char) 73) → I

o/p:
'hI! JOHN

(I3) i=2, 2<=7 (T)
ch = str.charAt(2) → '!'
ascii = (int)'!'
if (23 >= 65 && 23 <= 90)(F)
elseif (23 >= 97 && 23 <= 122) (F)
print((char)23) → !

Palendrome :

Method-1 :

Pentagon							
0	1	2	3	4	5	6	7

i → ← j
(i++) (j--)

P — n
e — o
n — g
t — a
o — t
g — n
o — e
n — P

Method-2 :

s1 → Pentagon
0 1 2 3 4 5 6 7
P → (i++)

s2 → nogatnep
0 1 2 3 4 5 6 7
j → ← j--

(j--) (i++)
s2(7) ← s1(0) P ← P
s2(6) ← s1(1) e ← e
:
s2(0) ← s1(7)

=> write a pgm to check if a string is palindrome (or) not.

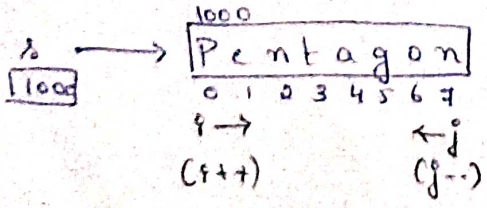
28/7/23

Palindrome → After reversing the string if we get same string back, then it is called palindrome.

Note: break → End the immediate loop.

continue → move to the next iteration of immediate loop

Pgm:
```
import java.util.Scanner;
class PalindromeDemo{
  P.S.V.M(){
    Scanner sc=new Scanner(System.in);
    S.O.Pln ("Enter the string: ");
    String s = sc.next();
    int i, j, flag=0;
    j = s.length()-1;
    for (i=0 ; i<= s.length()-1 ; i++)
    {
      if (s.charAt(i) != s.charAt(j))
      {
        flag = 1;
        break;
      }
      j--;
    }
    if (flag == 1) {
      S.O.Pln ("String is not a palindrome");
    }
    else {
      S.O.Pln ("String is a palindrome");
    }
  }
}
```



flag=0 → No mismatch till this point.

flag=1 → Mismatch found.

⇒ Write a pgm to check the frequency of occurance of a character in the given string.

output:

Enter the string: Pentagon

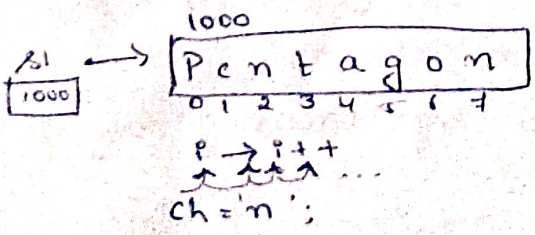
Enter character whose frequency must be checked: n ↵

Frequency of n = 2.

pgm:

```
import java.util.Scanner;
class CharacterFrequency{
    P.S.V.M(){
        Scanner sc= new Scanner(System.in);
        S.O.pln ("Enter the character whose frequency
                  is to be checked: ");
        char ch = (char) System.in.read();
                    (or)
        char ch = sc.next().charAt(0);
                  "n".charAt(0) → 'n'
        int i,count=0;
        for(i=0, ic=s.length()-1; i++)
        {
            if (ch == s.charAt(i))
            {
                count++;
            }
        }
        S.O.pln(" Frequency Of "+ch+" = "+count);
    }
}
```

S.O.pln("Enter the string: ");
String s = sc.next();



Note:

① Primitive Data == Primitive Data

- Here content will be compared.

② String == String
(object) (object)
(Non Primitive) (Non primitive)

- Here address will be compared.

Difference blw c-strings & Java-strings.

c-strings	Java-strings
• Here, the strings are an array of characters.	• Here, strings are object & not array of characters.
• C-strings end with null terminating character (\0)	• Java strings donot end with null terminating character.
• By default, c-strings are mutable.	• By default, Java-strings are Immutable. But mutable strings can be created as per requirement.
• C-strings are not secured.	• Immutable strings are highly secured.

String Tokenization:

- The process of splitting a string at specified characters is called string tokenization.
- The above can be achieved by creating object of StringTokenizer class.
- This class has 2 methods:
  ① hasMoreTokens() → returns true when tokens are present inside the object, & false otherwise.
  ② nextToken() → returns the next token in the object in string format.

⇒ Java pgm for to demonstrate String Tokenization.

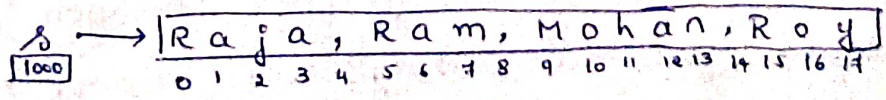
output:

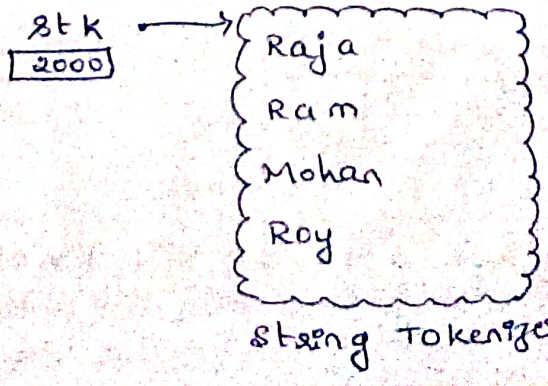
```
Enter a string : Raja, Ram, Mohan, Roy
Enter character where tokenization should take place: ,
After tokenization :
Raja
Ram
Mohan
Roy
```

pgm:

```
import java.util.*;
class TokenizeDemo{
  P.S.V.m(){
    Scanner sc = new Scanner (System.in);
    S.O.pln ("Enter the string :");
    String s = sc.next ();
    S.O.pln ("enter character where tokenization
          should take place :");
    char ch = (char) System.in.read();
    StringTokenizer stk = new StringTokenizer (s, ch);
    S.O.pln ("After tokenization :");
    String token;
    while(stk.hasMoreToken())
    {
      token = stk.nextToken();
      S.O.pln (token);
    }
  }
}
```

Exact syntax P.T.O.





Note: • character at which tokenization is done will not be a part of any of the tokens.

• StringTokenizer var = new StringTokenizer(String, String);

i) char ch = ',';
String s1 = "";
s1 = s1 + ch;
StringTokenizer stk = new StringTokenizer(s, s1);

ii) StringTokenizer stk = new StringTokenizer(s, String.valueOf(ch));

Mutable Strings:

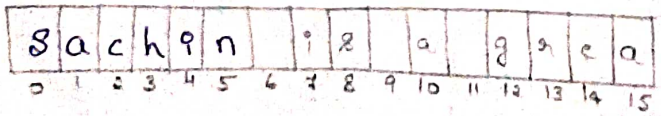
31/1/23

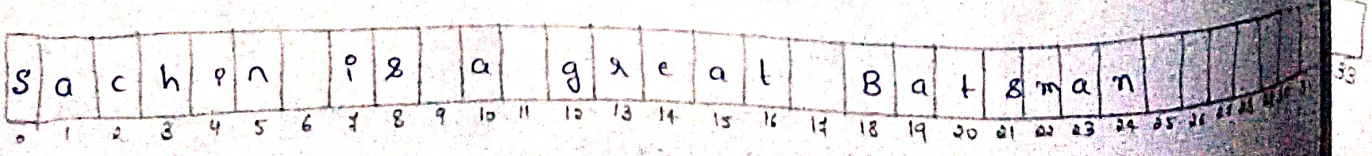
Difference b/w StringBuffer & StringBuilder class.

- StringBuffer is thread safe i.e only one thread can access it at a time.
- StringBuilder is not thread safe i.e multiple threads can access it at the same time.
- StringBuffer is less efficient when compare to StringBuilder
- StringBuilder is more efficient when compare to StringBuffer.

commands for creating mutable strings:

StringBuffer s1 = new StringBuffer("Pentagon");
StringBuilder s2 = new StringBuilder("Space");





Pgm:

```
class MutableDemo{
    p.s.v.m(){
        StringBuilder sl = new StringBuilder("Sachin");
        S.O.pln(sl.capacity());     // 22    (16+5)
        S.O.pln(sl.length());       // 6
        sl.append(" is a great Batsman");
        S.O.pln(sl.capacity());     // 46  (new capacity = 22*2+2 = 44+2)
        S.O.pln(sl.length());       // 25
    }
}
```

Note:

Default capacity = 16

New capacity = Existing capacity * 2 + 2

= ~~18~~ * ~~2~~ + ~~2~~ = ~~34~~.

- StringBuilder s = new StringBuilder();
  S.O.pln(s.capacity()); // 16 (16+0)

- StringBuilder s = new StringBuilder("Space");
  S.O.pln(s.capacity()); → 21 (16+5)

- StringBuilder s = new StringBuilder("Sachin");
  S.O.pln(s.capacity()); → 22 (16+6)

- StringBuilder s = new StringBuilder("Pentagon Space");
  S.O.pln(s.capacity()); → 29 (16+13).

- When an integer parameter is passed during creation of StringBuilder (or) StringBuffer, it is taken as default capacity.

- Other methods of StringBuilder & StringBuffer class.

i) capacity() ii) length() iii) append() viii) reverse()

iv) insert(int offset, String s)

v) delete(int start, int end)

vi) replace(int start, int end, String s)

vii) charAt(int index) [s[i] → direct indexing not allowed]

Methods in Java:

- A Method is a set of code placed within a class which gets executed only when it is invoked.
- Syntax:

```
returntype methodName (Parameter_List){

    // Executable statements

}
```

Note:

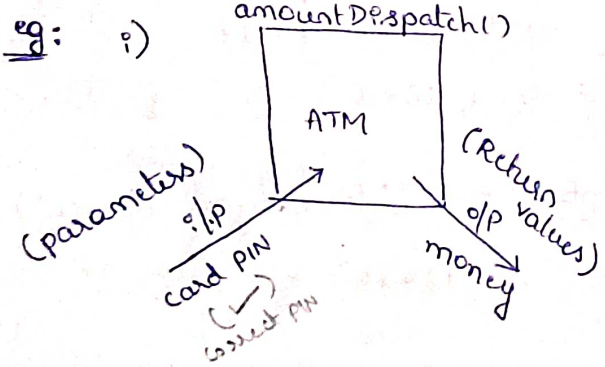
- In order to execute a method it must be called / invoked. otherwise the method will be in a dormant (or) inactive state.

Types of Methods in Java:

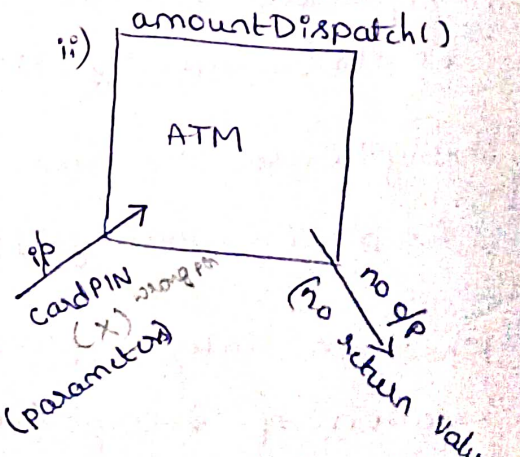
(ways / cases)

i) Methods with parameters with return values.
ii) Methods with parameters without Return values.
iii) Methods without parameters with return values.
iv) Methods without parameters without return values.

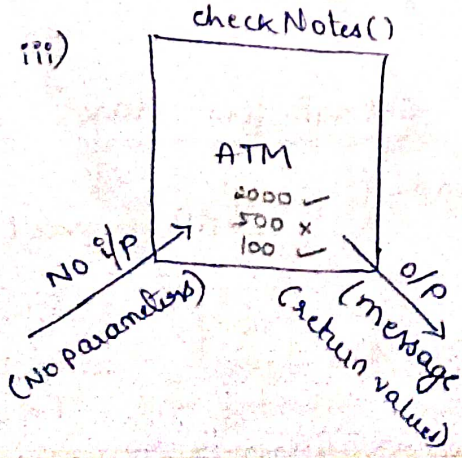
eg: i)



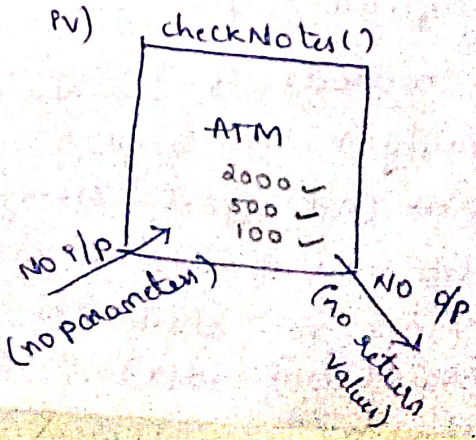
ii)



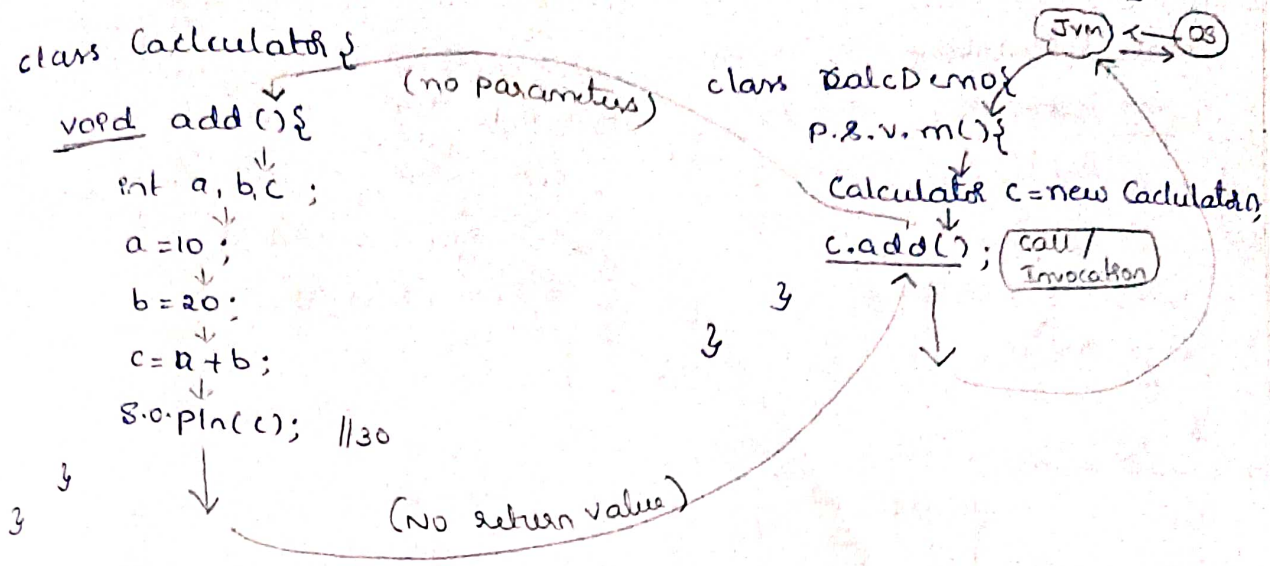
iii)

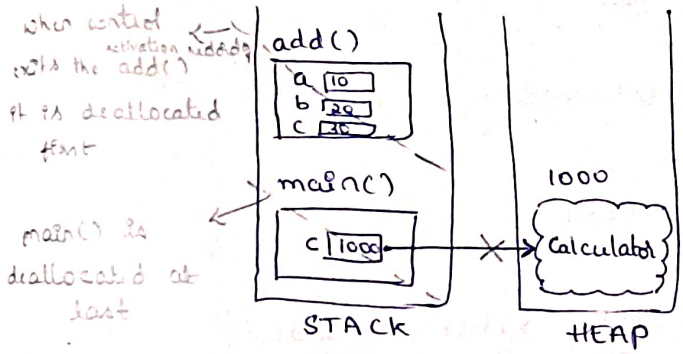


iv)

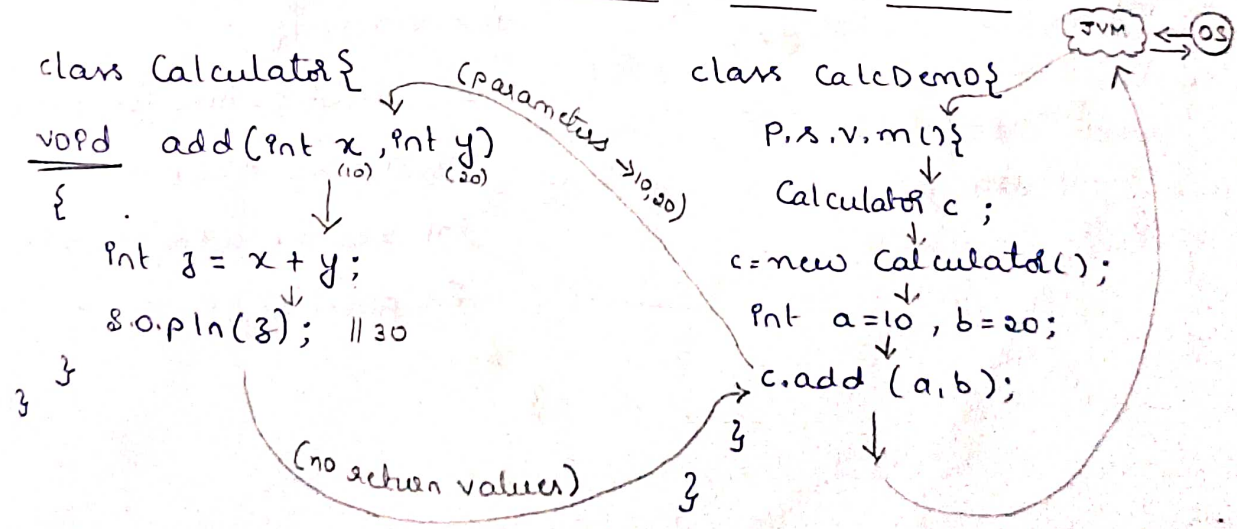


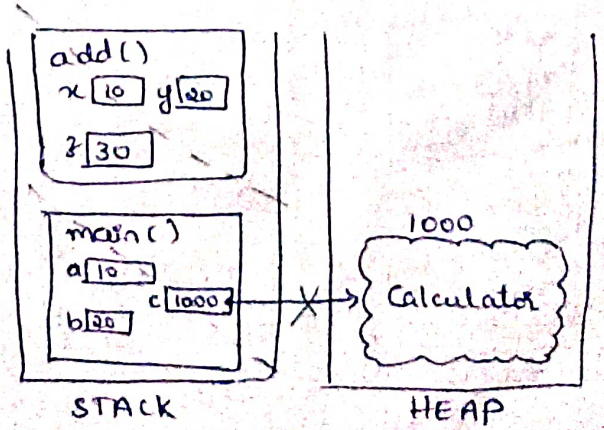
Methods without parameters without return values:





Methods with parameters without return values:





Methods without parameters with return values:

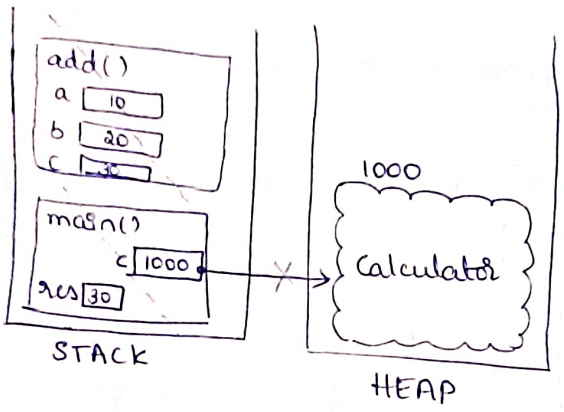
```
class Calculator{
    int add(){
        int a=10;
        int b=20;
        int c=a+b; //30
        return c;
    }
}
```

```
class CalcDemo{
    P.s.v.m(){
        Calculator c=new Calculator();
        int res=c.add();
        S.o.pln(res); //30
    }
}
```

JVM ⇄ OS

(no parameters)

(c = 30)



Methods with parameters with return values:

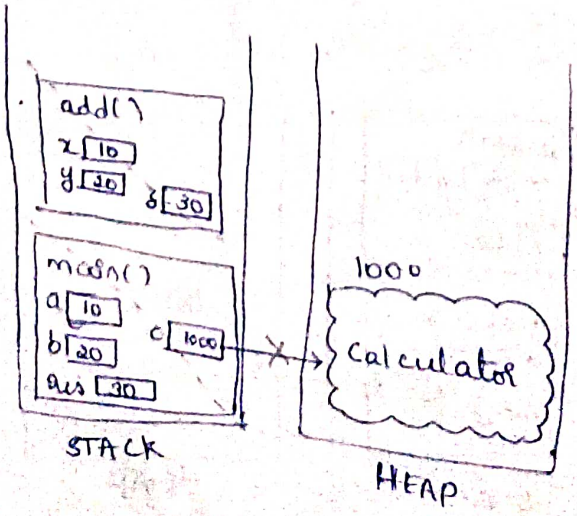
```
class Calculator{
    int add(int x, int y)
    {
        int z=x+y;
        return z;
    }
}
```

```
class CalcDemo{
    P.s.v.m(){
        Calculator c=new Calculator();
        int a=10, b=20;
        int res=c.add(a,b);
        S.o.pln(res); //30
    }
}
```

JVM ⇄ OS

(10, 20)

(30)



Note:

```
class Calculator {
    ? add () {
        int a = 10, b = 20;
        int r1, r2, r3, r4;
        r1 = a + b;
        r2 = a * b;
        r3 = a - b;
        r4 = a / 4;
        return r1, r2, r3, r4;
    }
}
```

→ Error

- Placing multiple values along with return statement produces an error in Java.
- If multiple values must be returned, they can be stored within an array / collection & send to the caller.
- In C when multiple values are placed with return, only last value will be sent back without any error.

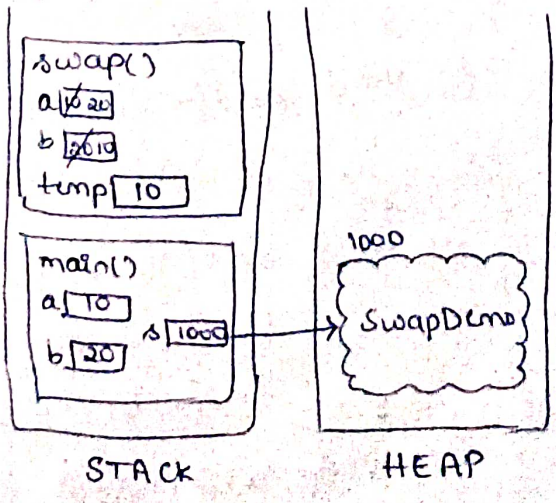
Analyze the following program:

```
class Demo {
    p.s.v.m () {
        int a = 10, b = 20;
        S.o.pln ("Before Swapping :");
        S.o.pln (a);   // 10
        S.o.pln (b);   // 20
        SwapDemo s = new SwapDemo();
        s.swap (a, b);
        S.o.pln ("After Swapping in main()");
        S.o.pln (a);   // 10
        S.o.pln (b);   // 20
    }
}
```

JVM ← OS

(10, 20)

```
class SwapDemo {
    void swap (int a, int b) {
        int temp;
        temp = a;
        a = b;
        b = temp;
        S.o.pln ("After swapping in swap()");
        S.o.pln (a);   // 20
        S.o.pln (b);   // 10
    }
}
```



	C	C++	Java
Pass by value	✓	✓	✓
Pass by ref	×	✓	×
pass by Address	✓	✓	×

Java supports only pass by value. Hence any changes made to the formal parameters will not effect the actual Parameters.

Method Overloading:

- The technique of writing multiple methods with the same name within same class is called method overloading
- conditions for method overloading:

* The methods intended for overloading must differ in atleast one of the following:
  - (a) Number of parameters.
  - (b) Type of parameters.
  - (c) Order of parameters.

- The method resolution also takes place in the above order.

* If all 3 are same for 2 methods then method resolution would not be possible & it will result an error.

- eg:

```
class Addition {
① void add ( ){
      int a =10, b=20;
      S.o.pln (a+b);
  }
② void add (int a){
      int b=20;
      S.o.pln (a+b);
  }
③ void add (int a + int b){
      S.o.pln (a+b);
  }
④ void add ( float a, float b){
      S.o.pln (a+b);
  }
⑤ void add (double a, double b){
      S.o.pln (a+b);
  }
⑥ void add ( int a, float b){
      float c=a+b;
      S.o.pln (c);
  }
⑦ void add ( float a, int b){
      float c=a+b;
      S.o.pln (c);
  }
```

```
(8) void add(int a, float b, double c){
        double d = a+b+c;
        S.O.pln (d);
    }
(9) void add (float a){
        float b=33.33f;
        s.o.pln (a+b);
    }
}
class AddDemo{
    p.s.v.m(){
        Addition a1 =new Addition();
        int a,b,c;
        float d, e, f;
        double g,h,i;
        a=10;
        b=20;
        c=30;
        d= 11.11 f;
        e = 22.22f;
        f = 33.33f;
        g=111.111;
        h =222.222;
        i = 333.333;

        a1.add();  // (1)
        a1.add(a);  (2)
        a1.add(a,b);  (3)
        a1.add(d, e);  (4)
        a1.add (g, h);  (5)
        a1.add (a, d, g);  (6)
        a1.add (a, d);  (7)
        a1.add (d, a);  (8)
    }
}
```

(5)
=> Why method Overloading is called as Virtual (or) False Polymorphism?

- Method overloading gives a perception that one method is performing multiple tasks. But in reality multiple methods with same name are performing individual tasks.

(6)
=> Why Method overloading is called as compile-time Polymorphism?

- out of several overloaded methods, Java decides which one to invoke at compile time i.e method resolution takes place even before execution. Hence M.O is called as compile-time polymorphism

(or) Static Binding (&) Early Binding.

4/8/23

Type casting:

- The process of converting one type to another type is called as type casting.
- There are 2 types of Type casting supported in Java.

i) Implicit Type casting:

- When the type casting is done automatically by compiler it is called as Implicit type-casting.
- eg:

```
int a;
byte b = 5;
a = (int) b   → Implicitly byte is converted into int
S.o.pln (a); // 5
```

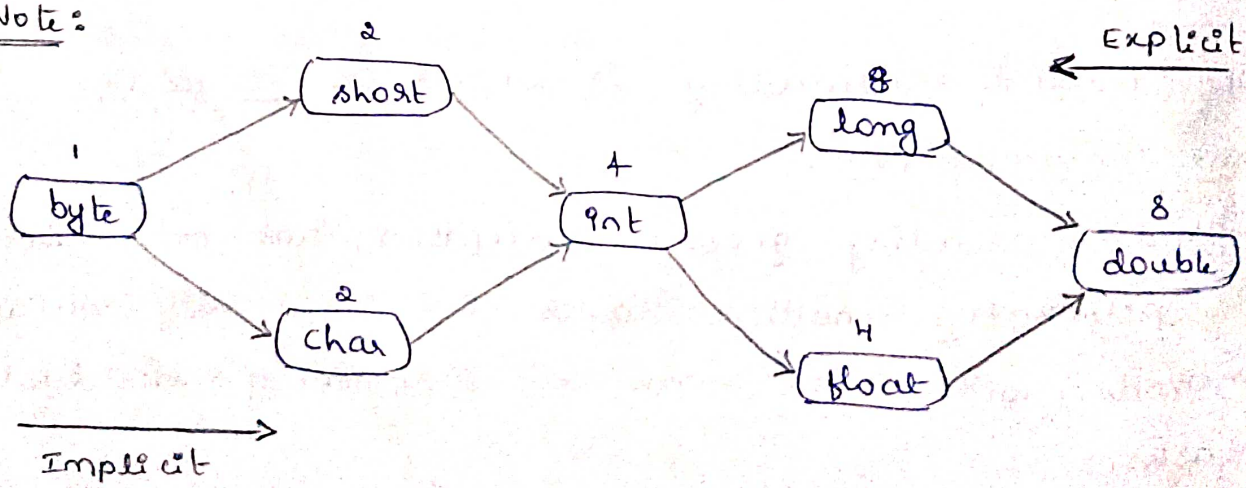
ii) Explicit Type casting:

When the type casting is done manually by the programmer, the process is called as Explicit Type casting.

eg:

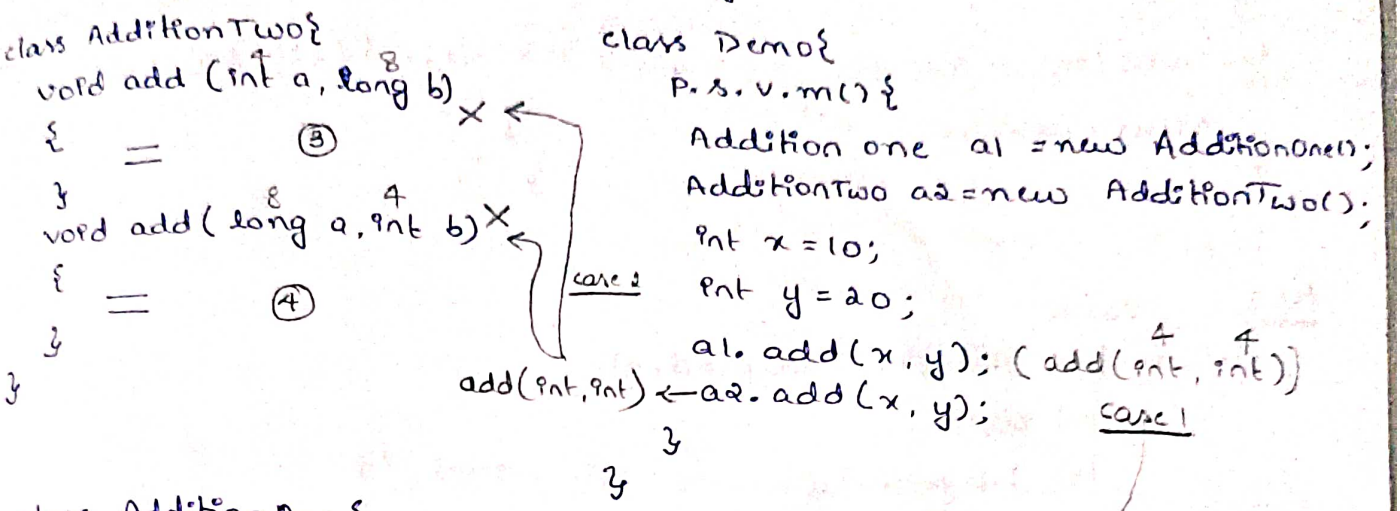
```
int a = 999999;
byte b;
b = (byte) a ;   → explicitly we need to convert int into byte
So.pln (b); // Data Loss / No Error
```

Note:



Numeric Type Promotion:

When resolving call to overloaded methods if in case resolution is not possible using number, type & order of parameters before giving error compiler will try to perform Numeric type promotion implicitly.



In case-1 ; (int, int) can be collected by (int, long) & (double, double) but since the nearest numeric type promotion is (int, long) → method ① will be called.

case-2 : (int, int) can be collected by both (int, long) & (long, int) But since both the numeric type promotions are nearest, method resolution is not possible due to ambiguity. Hence compile-time error is generated.

Q) => Is Method Overloading possible b/w two methods with the same signature but diff return type ?

↳ No. During method resolution we check number, type & order of parameters & if needed perform numeric type promotion. But we will never consider return type during

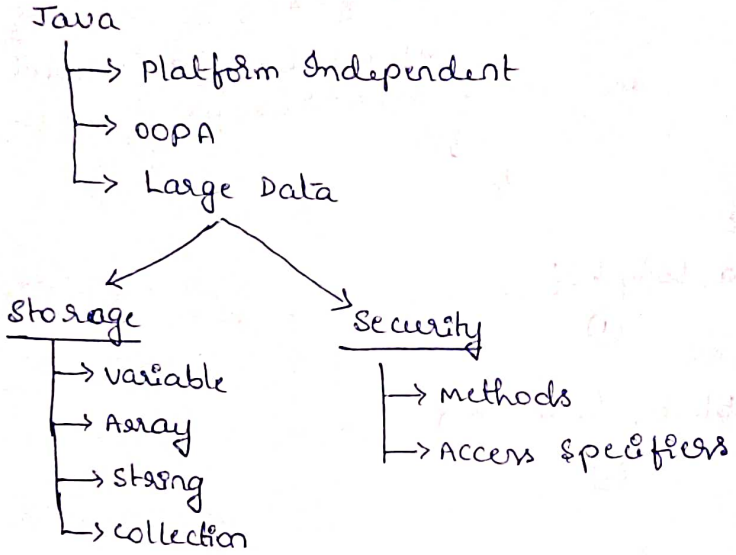
method resolution. Hence in the below eg. the compile time error is generated due to ambiguity.

```
class Addition {
    void add (int a, int b){
        S.o.pln(a+b);
    }
    int add (int a, int b){
        return (a+b);
    }
}
```

```
class Demo {
    P.S.V.m(){
        Addition a = new Addition();
        a.add(10, 20);
    }
}
```

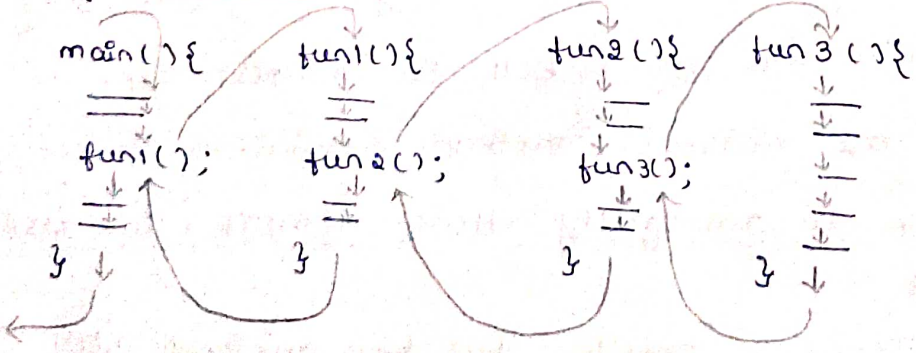
add(int, int)

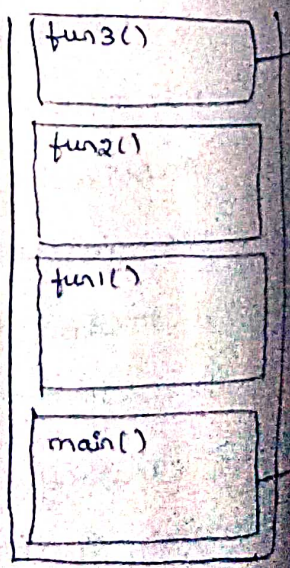
Note:



- As an Application level software, Java must be capable of storing & securing the large amount of data.

Why Activation Record must be created on the Stack?





- LIFO → Last in first out
  created last removed first.
- FILO → First in Last out
  created first removed last.

Encapsulation :

7/8/23

- It is a technique of providing controlled access to private members of a class using setters (mutator) & getters (Accessor).

- eg 1:

```
class Book{
    int pages;
}
```

No Security

```
class Demo {
    P.S.V.m(){
        Book b = new Book();
        b.pages = 100;
        S.O.pln(b.pages);   // 100
        b.pages = -100;
        S.O.pln(b.pages);   // -100
    }
}
```

In the above pgm, no security has been provided for pages which is an important part of class Book. Hence pages can be modified outside the class. This makes it insecure. In order to provide maximum security & ensure that no extanal entity can access pages, it has been marked private. Hence pages now can be accessed only within class Book.

```
class Book{
    Private int pages;
}
```

Too much security

```
class Demo{
    P.S.V.m(){
        Book b = new Book();
        b.pages = 100;      --> Error
        S.O.pln(b.pages);   Error
        b.pages = -100;     Error
        S.O.pln(b.pages);   Error
    }
}
```

In the above pgm, pages has been rendered useless. The idea of encapsulation is not to prohibit access but to provide restricted / controlled access to private members of a class. This can be achieved using accessors & mutators as shown below:

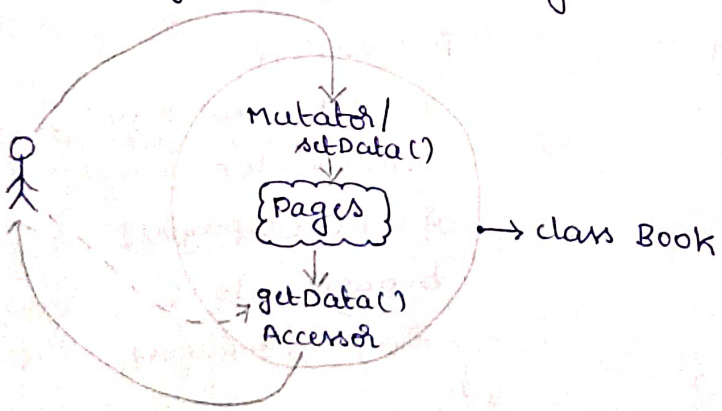
```
class Book{
  Private int pages;
  Public void setData(int x){        ← Setter/Mutator
    if(x>0){
      Pages = x;
    }
  }
  Public int getData(){              ← Getter/Accessor
    return pages;
  }
}
```

```
class Demo{
  P.S.V.m(){
    Book b=new Book();
    b.pages = 100; S.o.pln(b.pages);   ← Error
    b.setData(100);
    S.o.pln(b.getData()); // 100
    b.setData(-100);
    S.o.pln(b.getData()); // 0
  }
}
```

In the above pgm, we can see that access to pages is possible only through setData() & getData(). By employing proper verification & validation methods within setData() & getData(), appropriate security can be given to pages

Encapsulation is also called as Data Binding.

In the above eg, setData(), getData() & pages has been binding to form a single unit



Note:

this - Keyword (variable)

- It is a variable which holds the address of current object being used.
- It is an inbuilt variable which points to current

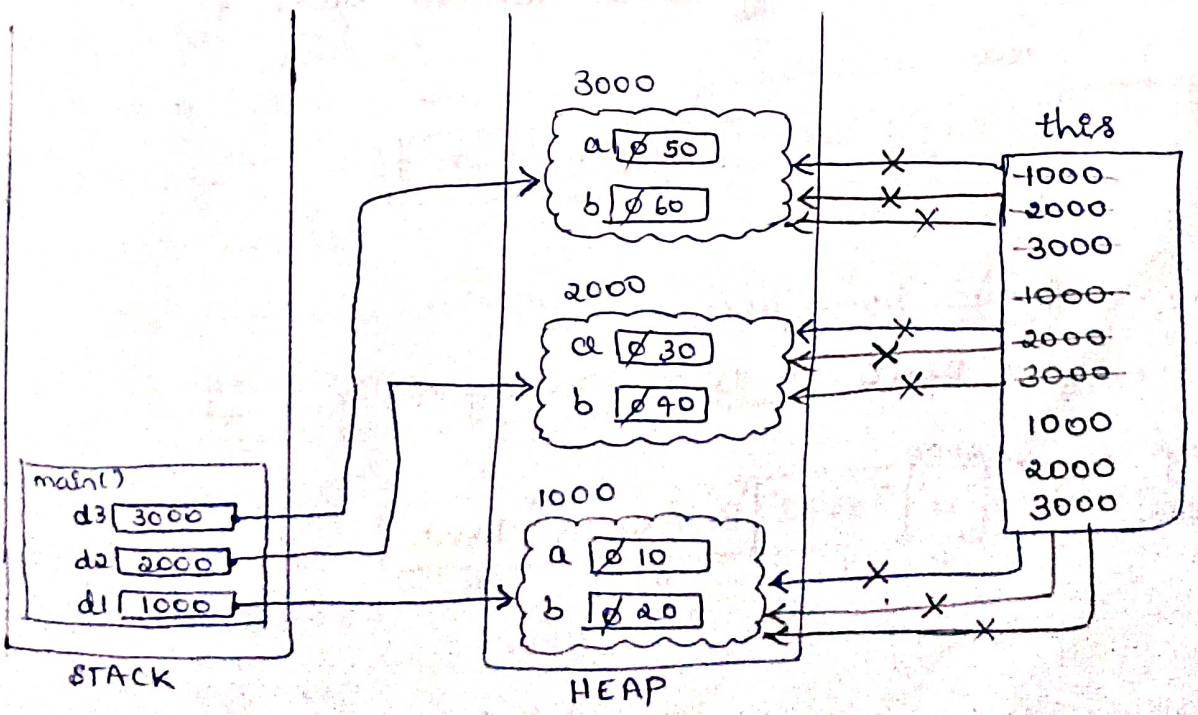
object along with address variable of same object.

- eg:

```
class Demo{
    int a,b;
}
```

```
class DemoApp{
    P.S.V.m(){
        Demo d1 = new Demo();
        Demo d2 = new Demo();
        Demo d3 = new Demo();
        d1.a = 10;
        d1.b = 20;
        d2.a = 30;
        d2.b = 40;
        d3.a = 50;
        d3.b = 60;
        S.o.pln (d1.a);
        S.o.pln (d2.b);
        S.o.pln (d2.a);
        S.o.pln (d2.b);
        S.o.pln (d3.a);
        S.o.pln (d3.b);
    }
}
```

JVM ⇄ OS



Command line Arguments:

- Arguments passed to main() method when it is called from command line (or) command prompt are called as command line arguments.
- These arguments are stored within an array of type string & address of this array is passed to main().

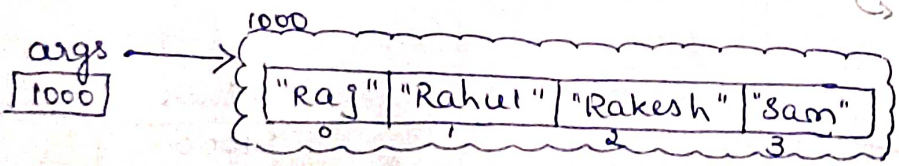
```
class Demo{
   P.S.V.m(String args[]){
        int i;
        for(i=0; i<=args.length-1; i++)
        {
          S.O.pln(args[i]);
        }
   }
}
```

— Demo.java

⇒ c:\...> javac Demo.java ↵

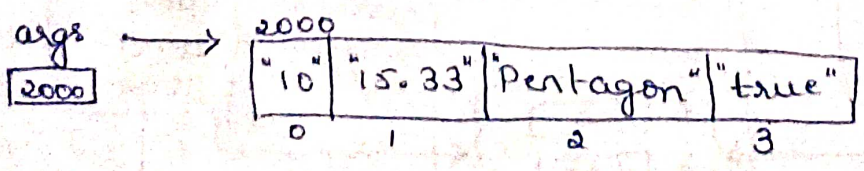
c:\...> java Demo Raj Rahul Rakesh Sam ↵

↳ command line arguments



⇒ c:\...> javac Demo.java ↵

c:\...> java Demo 10 15.33 Pentagon true ↵



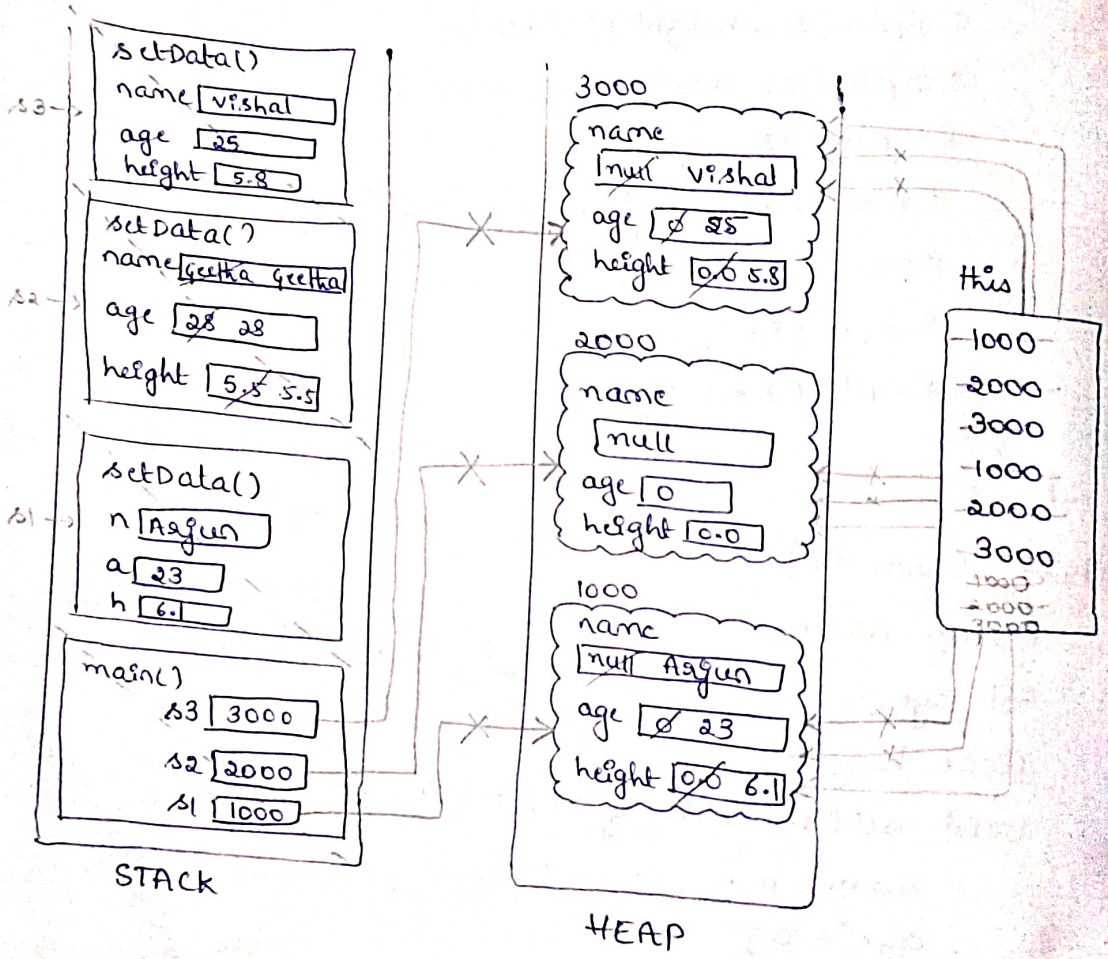
8/8/23

⇒ Write a note on Shadowing.

```
class StudentApp {
    p.s.v.m() {
        Student1 s1 = new Student1();
        Student2 s2 = new Student2();
        Student3 s3 = new Student3();
        s1.setData("Arjun", 23, 6.1f);
        s2.setData("Geetha", 28, 5.5f);
        s3.setData("Vishal", 25, 5.8f);
        S.o.println(s1.name);   // Arjun
        S.o.println(s1.age);    // 23
        S.o.println(s1.height); // 6.1f
        S.o.println(s2.name);   // null
        S.o.println(s2.age);    // 0
        S.o.println(s2.height); // 0.0
        S.o.println(s3.name);   // Vishal
        S.o.println(s3.age);    // 25
        S.o.println(s3.height); // 5.8
    }
}

class Student1 {
    String name;
    int age;
    float height;
    void setData(String n, int a, float h) {
        name = n;
        age = a;
        height = h;
    }
}
class Student2 {
    String name;
    int age;
    float height;
    void setData(String name, int age, float height) {
        name = name;
        age = age;
        height = height;
    }
}
```

```
class Student3{
    String name;
    int age;
    float height;
    void setData(String name, int age, float height){
        this.name = name;
        this.age = age;
        this.height = height;
    }
}
```

instance variable — local variable



- As per standards, the local variable assigned to instance variable must have the same name as the instance variable
- class Student1 dosnot follow this standard hence must be avoided.
- class Student2 follows the standard but as a result a name clash has occured.

whenever there is a name (same name) clash b/w local variable & instance variable, JVM always gives the preference to the local variable. This is called as shadowing.

- class student3 has resolved shadowing by using this keyword.

Constructors in Java:

A constructor is a special method that is called (invoked) and executed during object creation.

properties of a constructor:

i) It has the same name as that of the class.

ii) It doesnot have a return type.

iii) Note: If return type is provided to a constructor, it is no longer a constructor & acts as a normal method.

```
class student{
    String name;
    int age;
    float height;
    void setData(String name, int age, float height){
        this.name = name;
        this.age = age;
        this.height = height;
    }
}
```

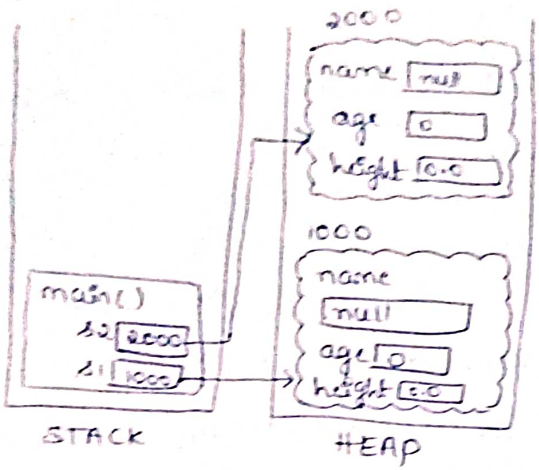
↑ Before compilation →

During compilation →

compiler (javac)

```
class student{
    String name;
    int age;
    float height;
    student(){                      } special method/
        name = null;                  constructor/
        age = 0;                      Default constructor
    } height = 0.0f;                }
    void setData(String name, int age, float height){
        this.name = name;               this.height = height;
    } this.age = age;
}
```

```
class StudentApp{
    P.S.V.m(){
        Student s1 = new Student();
        Student s0 = new Student();
    }
}
```



- Calling the constructor explicitly after object construction like s1.Student() (or) s0.Student() results an error.
- In the above pgm, the objects created are in an invalid state. We can bring it to a valid state by calling setData() method. But as per standards, when object is created then itself it must be in valid state.
- In order to create an object in valid state we must write a parameterized constructor.

Note:

- Compiler will insert a default / 0(zero) parameter constructor only if the programmer has not written his/her own prog.no constructor.
- When there are more than one constructors in the same class we call it as constructor overloading.
- Constructor is called only once during object creation. Any changes to be made to the object after this

process must be done with the help of mutator (setData)

Updated steps of Object creation:

i) creat address variable.
ii) using new allocate memory for object.
iii) Allocate memory for instance variables within object.
iv) call the & execute constructor of the object.
v) collect address returned by new.

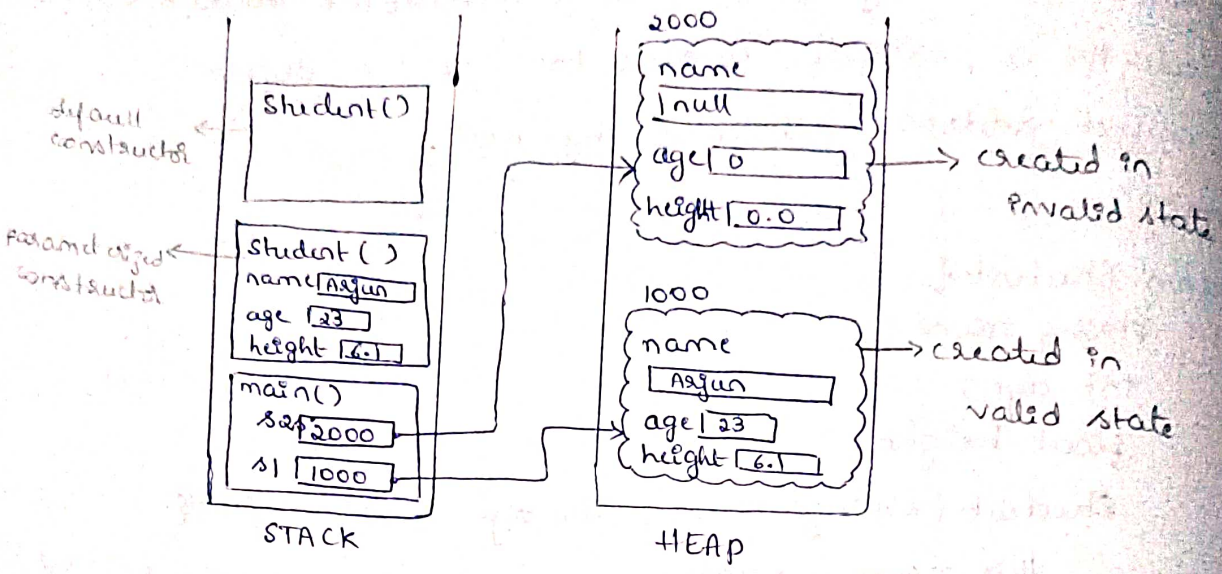
pgm:

```
class Student{
    String name;
    int age;
    float height;
    Student(String name, int age, float height){
        this.name = name;
        this.age = age;
        this.height = height;
    }
    Student(){
        name = null;
        age = 0;
        height = 0.0f;
    }
    void setData(String name, int age, float height){
        this.name = name;
        this.age = age;
        this.height = height;
    }
}
class StudentApp{
    P.S.V.m(){
        Student s1 = new Student("Arjun", 23, 6.1f);
        Student s2 = new Student();
        S.O.pln(s1.name);   // Arjun
        S.O.pln(s1.age);    // 23
        S.O.pln(s1.height); // 6.1
```

Parameterized constructor

Zero parameter constructor

```
        S.O.pln (s2.name);    // null
        S.O.pln (s2.age);     // 0
        S.O.pln (s2.height);  // 0.0
    }
}
```



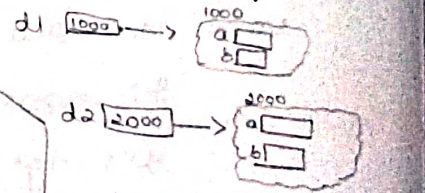
Static keyword :

9/8/23

(I) For static variable memory will be allocated only once

- Variables marked as static will be allocated memory only once on the static space. irrespective of no of objects created.
- The no of instance variables in memory is equal to the no of objects created.

```
class Demo {
    int a, b;
    static int x, y;
}
class DemoApp {
    public void main(String args[]){
        Demo d1 = new Demo ();
        Demo d2 = new Demo ();
        Demo d3 = new Demo ();
    }
}
```



object created = 2 time
no of instance variable
↓
a = 2 time
b = 2 time

(-1)

Note: static → To make the variable common
final → To make the variable constant

CODE SEGMENT

```
class Demo {
    Int a, b
    static int x, y;
}
class DemoApp {
    P.V.m() {
        Demo d1 = new Demo();
        Demo d2 = new Demo();
        Demo d3 = new Demo();
    }
}
```

STACK SEGMENT

```
P.V.main() {
    Demo d1 = new Demo();
    Demo d2 = new Demo();
    Demo d3 = new Demo();
}
```

(4)

JVM → class loader (2) (3)

* Static variables

STATIC SPACE

x

y

OBJECT SPACE (Heap)

d3 | 3000 → 3000 { a 0, b 0 }

d2 | 2000 → 2000 { a 0, b 0 }

d1 | 1000 → 1000 { a 0, b 0 }

RAM

(11)

- Instance variables are specific to the object & hence they can be accessed only through the corresponding address variable.
- Static variables on the other hand are shared among all the objects. Hence they can be accessed using any address variable of the class. This makes static variables specific to the class & not to the object.
- Hence static variables are also called as class variables.

x, y → accessed by d1, d2, d3

d1 → 0 10 20

```
CODE SEGMENT
class Demo{
    int a, b;
    static int x, y;
}
class Demo1{
    P.V.m(){
        Demo d1=new Demo();
        Demo d2=new Demo();
        Demo d3=new Demo();
        d1.a=10;
        d2.b=20;
        d1.x=9;
        d1.y=19;
        S.o.pln(d1.a); //10
        S.o.pln(d1.b); //20
        S.o.pln(d1.x); //9
        S.o.pln(d1.y); //19
        d2.a=10;
        d2.b=20;
        d2.x=99;
        d2.x=109;
        S.o.pln(d2.a); //10
        S.o.pln(d2.b); //20
        S.o.pln(d2.x); //99
        S.o.pln(d2.y); //109
        d3.a=10;
        d3.b=20;
        d3.x=999;
        d3.y=1009;
        S.o.pln(d3.a); //10
        S.o.pln(d3.b); //20
        S.o.pln(d3.x); //999
        S.o.pln(d3.y); //1009
    }
}
```

STACK SEGMENT

d3 3000 → 3000: a 10, b 20

d2 2000 → 2000: a 10, b 20

d1 1000 → 1000: a 10, b 20

STATIC SPACE

x 999

y 1009

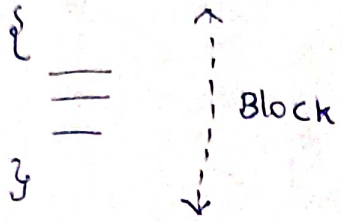
JVM → class loader → * static variables

RAM

Blocks in Java:

- A set of code / statements enclosed within flower brackets is called Block.
- The nature of the block changes as we associate it with different keywords.

eg:



if-Block:

```
if()
{
   ___
   ___
   ___
}
```

for-Block:

```
for( )
{
   ___
   ___
   ___
}
```

static Block:

```
static
{
   ___
   ___
   ___
}
```

Non-static Block:

```
{
   ___
   ___
   ___
}
```

(iii)

⇒ • The static block will be loaded to the static space by the class loader & immediately executed.

- In the presence of non-static block it will be executed before execution of constructor.
- Static block will be loaded to the stack from static space.
- Non-static block will be loaded to the stack from code segment.

CODE SEGMENT

```
class Demo{
   static Pnt x, y;
          Pnt a,b;

   static
   {
    S.o.pln("Inside static Block");
   }

   {
    S.o.pln("Inside Non-static
                    block");
   }

   Demo(){
     S.o.pln("Inside Demo constructor");
   }
}
class StaticDemo{
     P.v.m(){
       Demo d1;
         d1=new Demo();
     }
}
```

STATIC SEGMENT

```
Demo(){ (3)
  S.o.pln("Inside Demo Constructor");
}                                        3rd

{ (2)
 S.o.pln("Inside Non-static block");
}                                        2nd

main(){                                  4th
   Demo d1;
   d1 =new Demo();
}

static { output → (1)
  S.o.pln("Inside static block");
}                                        terminated → 1st
```

JVM ⟶ class loader

* static variables
* static Blocks

5th

STATIC SPACE

x y

```
static{
  S.o.pln("Inside static
           block");
}
```

OBJECT SPACE

d1 | 1000 → 1000 { a, b }

RAM

CODE SEGMENT

```
class Demo{
 int a,b;
 static int x,y;
 Demo(){
  S.o.pln("Inside Demo constructor");
 }
 static {
  S.o.pln("Inside static block");
 }
 {
  S.o.pln("Inside non-static block");
 }
 void nonStaticMethod(){
  S.o.pln("Inside non-static method");
 }
static void staticMethod(){
  S.o.pln("Inside static method");
 }
}
 class StaticDemo{
       P.v.m(){
        Demo d1=new Demo();
        d1.nonStaticMethod();
        d1.staticMethod();
       }
 }
```

STACK SEGMENT

```
static void staticMethod(){                    X5
  S.o.pln("Inside static method");   (5)
}

void nonStaticMethod(){                        X4
  S.o.pln("Inside non-static method");
}                                  (4)

Demo(){                                        X3
  S.o.pln("Inside demo constructor");
}                                  (3)

{                                              X2
  S.o.pln("Inside non-static block");
}                                  (2)

main(){                                        X6
  P.v.m(){
    Demo d1 = new Demo();
    d1.nonStaticMethod();
    d1.staticMethod();
  }

static {                                       Terminated(X1)
  S.o.pln("Inside static block");
}                                  output -(1)

JVM ⟶ class loader                             X7
```

STATIC SPACE

```
x [ ]   y [ ]

static {
  S.o.pln("Inside static block");
}

static void staticMethod(){
  S.o.pln("Inside static method");
}
```

OBJECT SPACE

d1 [1000] → 1000 { a [ ], b [ ] }

* Static variables
* Static Blocks
* Static Methods

(IV)

- class loader loads static method on to the static space but unlike static block it is not executed immediately.
- Static method gets executed only when it is called.

Steps Involved in execution of Java pgm:

i) pgm to be executed is loaded on to code segment.

ii) JVM gets loaded on to the stack.

iii) JVM transfers control to class loader which loads static variables, static blocks & static methods onto the static space.

iv) control given back to JVM which continues execution of the pgm from main().

Two ways of calling Static Methods:

i) using address variable.

ii) using class name.

```
class Demo{
    int a, b;
    static int x, y;
    void nonStaticMethod(){
        S.o.pln("Inside non static method");
    }
    static void staticMethod(){
        S.o.pln("Inside static method");
    }
}
class StaticDemo{
    P.S.V.m(){
        Demo d1=new Demo();
        d1.a = 10;
        d1.b = 20;
        d1.Demo.x =99;
        Demo.y = 109;
        S.o.pln(d1.a); //10
        S.o.pln(d1.b); //20
        S.o.pln(Demo.x); //99
        S.o.pln(Demo.y); //109
        d1.nonStaticMethod();
        // Inside non static method
        Demo.staticMethod();
    } //Inside static method
}
```

*** A static method can be called even without creating an object of the class.

(using class name (Demo.x))

(*) => Why main() method must be marked as static ?

- It is the JVM that makes a call to main().
- But this call must be made without creating an object of the class. Inside which main() is present.
- This is only possible if main() is marked as static
- As per standards, file name of a pgm must be same as class name within which main() is present so that, JVM can recognize where main() is present out of many class in the pgm.

Q ⇒ Is it compulsory for file name to be same as class name within which main() is present?

↳ No. It is a standard to avoid confusion. Any name can be given as file name. But during execution name of the class within which main() is pres. must be specified.

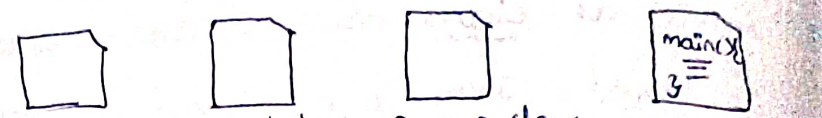
eg:

```
class Demo{
    ==
}
class Demo1{
    ==
}
class Demo2{
    ==
}
class Demo3{
    P.S.V.main(){
    ==
    }
}
```

Pentagon.java

cmd prompt:

C:\...> javac Pentagon.java ↵ ✓



Demo.class Demo1.class Demo2.class Demo3.class

C:\...> java Pentagon ↵ ✗

C:\...> java Demo ↵ ✗

C:\...> java Demo1 ↵ ✗

C:\...> java Demo2 ↵ ✗

C:\...> java Demo3 ↵ ✓

Note: Non-static block gets executed each time an object is created. whereas, static blocks gets executed only once per class. If there are multiple blocks, they get executed in the order in which they are written.

- Static block will be executed only once per class but only if atleast one object of that class is creat

eg:

```
class Demo {
    static {
        S.O.Pln("SB 1");
    }
    static {
        S.O.Pln("SB 2");
    }
    static {
        S.O.Pln("SB 3");
    }
    {
        S.O.Pln("NSB 1");
    }
    {
        S.O.Pln("NSB 2");
    }
    {
        S.O.Pln("NSB 3");
    }
}
```

```
class StaticDemo{
    p.s.v.m(){
        Demo d1 = new Demo();
        Demo d2 = new Demo();
        Demo d3 = new Demo();
    }
}
```

o/p:
```
SB 1
SB 2      } static block executed only once
SB 3
NSB 1
NSB 2
NSB 3     no of NON-static blocks executed = no of times object created
NSB 1
NSB 2
NSB 3
NSB 1
NSB 2
NSB 3
```

Rules for accessing Instance Variables & Static Variables:

A variable can be accessed if & only if it is already allocated memory.

i) static variables can be accessed within static block.

ii) static variables can be accessed within static method.

iii) Instance variables can be accessed within non-static block.

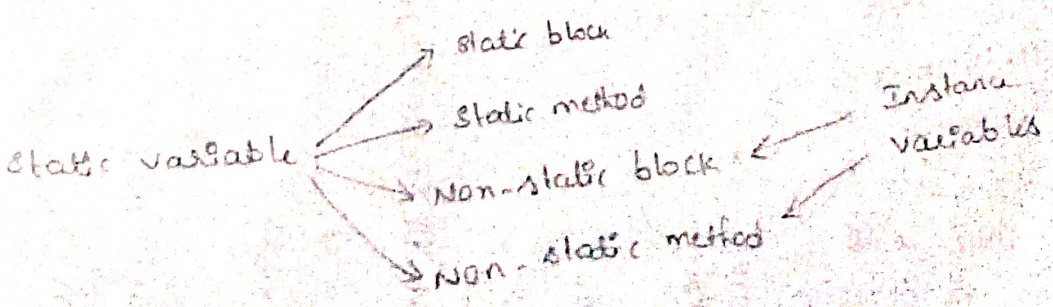
iv) (I.V) can be accessed within non-static method.

v) static variables can be accessed within non-static block.

vi) static variables can be accessed within non-static method.

vii) Instance variables cannot be accessed within static block.

viii) (I.V) cannot be accessed within static method.



CODE SEGMENT

```
class Demo{
    int a,b;
    static int x,y;
    static {
        a=10;
        b=20;
    }
    {
        a=30;
        b=40;
    }
static void staticMethod(){
    S.o.pln(a);
    S.o.pln(b);
}
class StaticDemo{
    P.S.V.m(){
        Demo.staticMethod();
    }
}
```

STACK SEGMENT

X_2

```
static staticMethod(){
    S.o.pln(a);   → Error
    S.o.pln(b);   → Error
}
```

X_3

```
main(){
    Demo.staticMethod();
}
```

X_1

```
static {
    a=10;   → Error
    b=20;   → Error
}
```

X_4

JVM → class loader

- * Static variables
- * Static Blocks
- * Static Methods

STATIC SPACE

x

y

```
static {
    a=10;
    b=20;
}
```

```
staticMethod(){
    S.o.pln(a);
    S.o.pln(b);
}
main(){
    ==
}
```

Note: i)
```
class Demo{
    int a, b, c;
    {
    a=10; b=20;
    }
}
```

```
class staticDemo{
    P.S.V.m(){
    S.o.pln Demo d1 = new Demo();
     S.o.pln (d1.a);  // 10
     S.o.pln (d1.b);  // 20
     S.o.pln (d1.c);  // 0
    }
}
```

ii)
```
class Demo{
    int a, b, c;
    {
    a=10; b=20;
    }
    Demo(){
        a=0;
        b=0;
    }
}
```

```
class static Demo{
    P.S.V.m(){
    Demo d1 = new Demo();
     S.o.pln (d1.a);  // 0
     S.o.pln (d1.b);  // 0
     S.o.pln (d1.c);  // 0
    }
}
```

=> (1) eg for <u>appln</u> of <u>static block</u>:

```
class Addition {
    static int a, b, c, d;
    static {
        a = 0;
        b = 0;
        c = 0;
        d = 0;
    }
    void add(){
        S.o.pln(a+b+c+d);
    }
    void add(int x){
        a = x;
        S.o.pln(a+b+c+d);
    }
    void add(int x, int y){
        a = x;
        b = y;
        S.o.pln(a+b+c+d);
    }
    void add(int x, int y, int z){
        a = x;
        b = y;
        c = z;
        S.o.pln(a+b+c+d);
    }
    void add(int x, int y, int z,
                int q){
        a = x;
        b = y;
        c = z;
        d = q;
    }
}
```

```
class AdditionDemo{
    P.S.V.m(){
    Addition a1 = new Addition();
    a1.add();          // 0
    a1.add(10);        // 10
    a1.add(10,20);     // 30
    a1.add(10,20,30);  // 60
    a1.add(10,20,30,40); // 100
    }
}
```

The appn of static bloc

- The appln of static blocks is to initialize static variables.

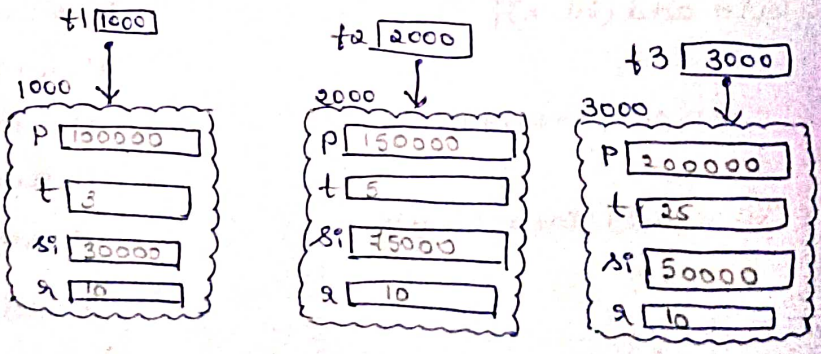
(2) The appln of static methods is to use methods even without using object of the class.

(3) The appln of static variables :

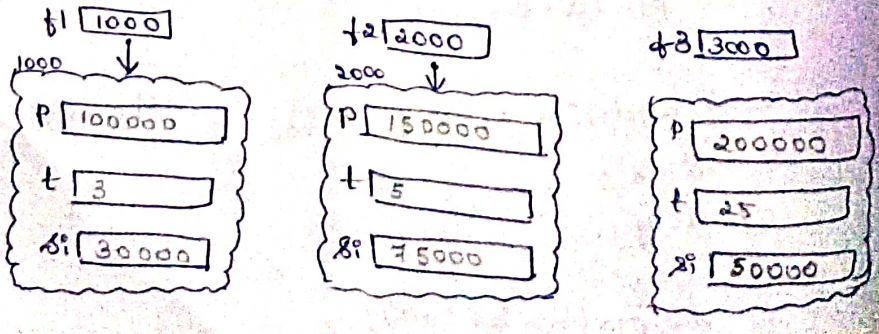
```
import java.util.Scanner;
class Farmer{
 float p, t, si;
 static float r;
 void input(){
  Scanner sc=new Scanner(System.in);
  S.o.pln("Enter the principal:");
  p= sc.nextFloat();
  S.o.pln("Enter the time: ");
  t=sc.nextFloat();
 (r=10.0f;) ⟶ static{ r=10.0f; }
 }
 void calcSi(){
  si=(p*t*r)/100;
 }
 void disp(){
  S.o.pln("SI ="+si);
 }
}
```

```
class FarmerApp{
 P.S.V.m(){
  Farmer f1=new Farmer();
  f1.input();
  f1.CalcSi();
  f1.disp();
  Farmer f2=new Farmer();
  f2.input();
  f2.CalcSi();
  f2.disp();
  Farmer f3=new Farmer();
  f3.input();
  f3.CalcSi();
  f3.disp();
 }
}
```

↳ Without static keyword



↳ With static keyword



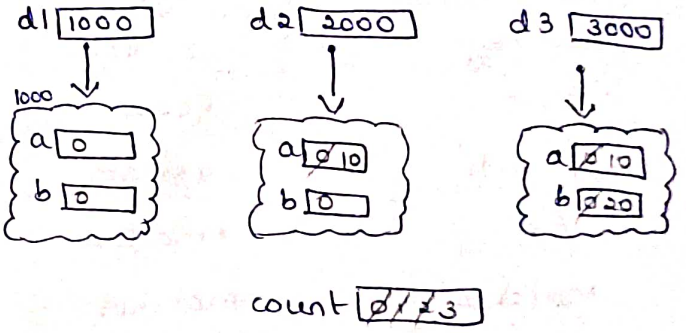
- While designing the appln, values that are common to all the objects created must be marked as static so that memory is allocated only once.
- In the above pgm, rate of interest (r) is common for all the objects. Hence it must be marked as static.

(7) Appln of non-static block:

- To initialize non-static variables (Instance variables)
- To count the no of objects created.

```
class Demo{
 int a, b;
 static int count;
 Demo(){
 }
 Demo (int x){
   a = x;
 }
 Demo(int x, int y){
   a = x;
   b = y;
 }
 {
   count ++;
 }
}
```

```
class DemoApp {
   P.S.V.m(){
     Demo d1 = new Demo();
     Demo d2 = new Demo();
     Demo d3 = new Demo();
     S.o.pln(Demo.count); // 3
   }
}
```



- Default values of static variables is same as that of Instance variables.

(8) List the diff b/w Local, Instance & Static variables:

(already written)

⇒ Static variables:

- Declared within class outside the methods.
- Default values depends on data type, it is not garbage
- Allocated memory on the static space.
- Allocated memory when the execution of pgm begins.
- Deallocated memory when the pgm completes its execution.

Note:

- The part of memory where a variable is allocated is called as Scope.
- There are only 3 scopes in Java → Local, Instance & Static scope.

Inheritance:

- The object oriented technique using which the features of one class (properties & behaviours) is inherited into another class. is called as Inheritance.
- With the introduction of Inheritance, the development time of projects reduced significantly as the same existing code could be re-used.

Eg: Profit(↑) Time(↓)

xyz

Name
Age
country
Runs
Wickets
6's
Matches

Name
Age
country
Goals
Assists
Penalties
Matches

Without Inheritance

ABC

Name
Age
country
Matches

Runs
Wickets
6's

Goals
Assists
Wickets

With Inheritance

→
```
class A{
    int i;
}
class B{
    int j;
}
```
unrelated classes

- In order to establish a relationship b/w 2 classes, the extends keyword must be used.

```
class A{
    int i;
}
class B extends A{
    int j;
}
```
A: Parent / Super / Base class

B: child / Sub / Derived class

- When an object of child class is created, the parent class constructor is executed before child class constructor.

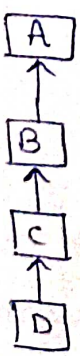
```
class A {
    A() {
        S.o.pln("A's constructor");
    }
}
class B extends A {
    B() {
        S.o.pln("B's constructor");
    }
}
class X {
    X() {
        S.o.pln("X's constructor");
    }
}
class Y extends X {
    Y() {
        S.o.pln("Y's constructor");
    }
}
class Z extends Y {
    Z() {
        S.o.pln("Z's constructor");
    }
}
```

```
class Demo {
    P.S.V.m() {
        B b = new B();  // A's constructor
                        B's constructor
        A a = new A();  // A's constructor
        Z z = new Z();  // X's constructor
                        Y's constructor
                        Z's constructor
        Y y = new Y();  // X's constructor
                        Y's constructor
        X x = new X();  // X's constructor
    }
}
```

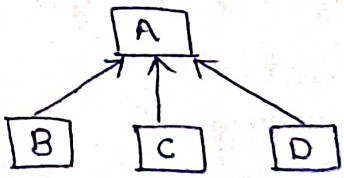
Types of Inheritance in Java:

i) Single (or) Inheritance:

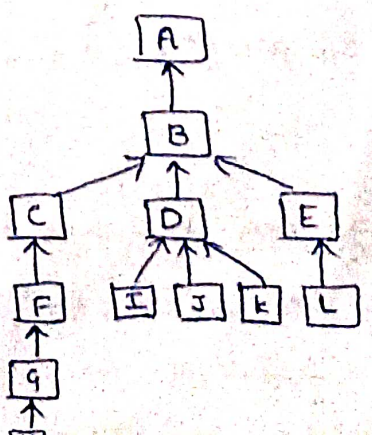
ii) Multilevel Inheritance:



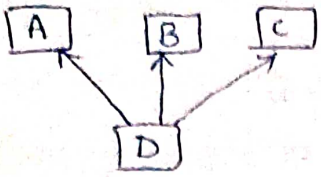
iii) Hierarchical Inheritance:



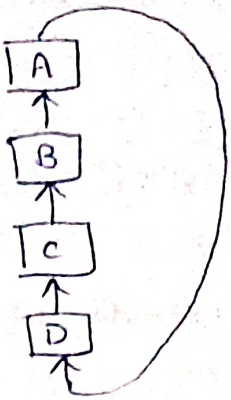
iv) Hybrid Inheritance:



v) Multiple Inheritance : (Not allowed)



vi) Cyclic Inheritance :



- Multiple Inheritance cannot be achieved in Java using classes as it leads to diamond-shaped pblm.
- But it can be achieved using Interfaces.
- There is no way of achieving cyclic inheritance in Java.

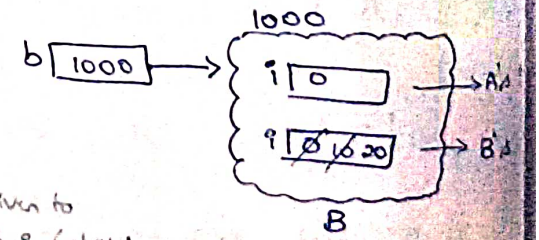
(Q) Explain Hiding.

- When a name clash occurs b/w the instance variable of parent class & instance variable of child class, preference is given to instance variable of child class. This is called as Hiding.

```
class A {
    int i;
}
class B extends A {
    int i;
    int i;
    B(int x, int y){
        i=x;
        i=y;
    }
    void DISP (){
        S.o.pln ("A's i =" + i); //20
        S.o.pln ("B's i =" + i); //20
    }
}
```

Here, Preference will be given to B's i (child class)

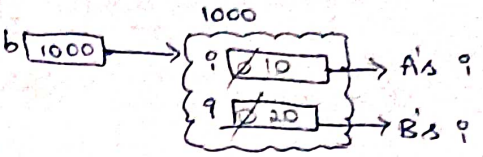
```
class Demo{
    P.S.V.m(){
        B b=new B(10,20);
        b.disp();
    }
}
```



- Hiding can be resolved using super keyword.
- super is used to refer to immediate parent class.

```
class A{
  int i;
}
class B extends A{
  int i;
  B(int x, int y){
    super.i = x;   (A's i)
    i = y;         (B's i)
  }
  void disp(){
    S.o.pln("A's i="+super.i);  //10
    S.o.pln("B's i="+i);        //20
  }
}
```

```
class Demo{
  P.S.V.m(){
    B b=new B(10,20);
    b.disp();
  }
}
```



⇒ Analyze the following pgm.

17/8/23

```
class Room{
  int l, b;
  Room(int x, int y){
    (1) l = x;
    (2) b = y;
  }
  void area(){
    S.o.pln(l*b);
  }
}
```

```
class LivingRoom extends Room{
  int l, b;   → It is taken automatically from Room class. No need to write again.
  int h;
  LivingRoom(int x, int y, int z)
  {
    super(x, y);
    (3) h = z;
  }
  void volume(){
    S.o.pln(l*b*h);
  }
}
```

```
class Demo{
  P.S.V.m(){
    LivingRoom lr=new LivingRoom(10,20,30);
    lr.area();     //200
    lr.volume();   //6000
  }
}
```

- We know that parent constructor is executed before child constructor. But unless specified the control always looks for default constructor of parent class.

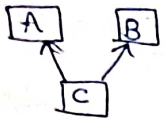
- In the above eg, parent Room doesnot have a default constructor. Hence we must explicitly place a call to parameterized constructor of parent room. This can be achieved using super keyword.

- super(parameter_list) must be the very first line in the constructor of child class.

Note:

i) class A{
  int i;
}
class B{
  int i;
}
class C extends A, B{ → Error
  int i, int i;
  int i;
}

(multiple)



- class C has a 2 immediate parents A, B. Using super we can refer to the immediate parents
- But as there are 2 immediate parents (A, B) in this case, it leads to ambiguity.

ii) class A{
  int i;
}
class B extends A{
  int i;
  int i;
}
class C extends B{
  int i; int i;
  int i;
}

(c)

- From the child class using super we cannot refer to Grandparent (A) & beyond.
- In this pgm, using only i we can access C's i, using super.i we can access B's i, but A's i cannot be accessed.

⇒ Eg for Application of Inheritance:

* (Without using Inheritance)

class PassengerPlane{
  void takeOff(){
    S.o.pln("Plane took off");
  }
  void fly(){
    S.o.pln("Plane is flying");
  }
  void carryPassenger(){
    S.o.pln("Carrying Passenger");
  }
  void land(){
    S.o.pln("Plane has landed");
  }
}

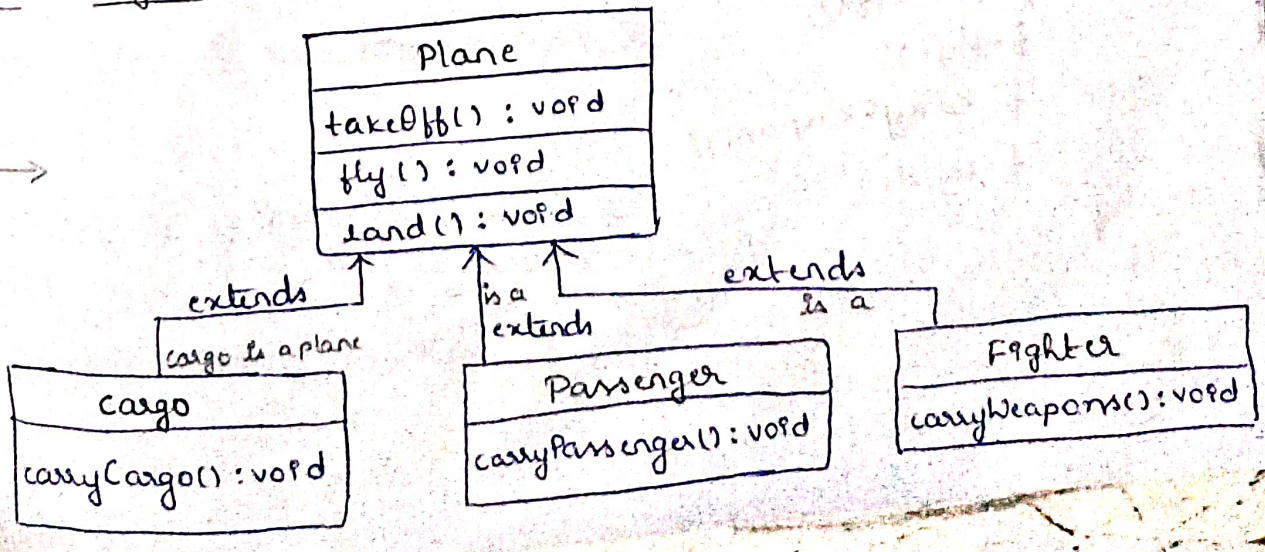
```
class CargoPlane{
    void takeOff(){
        S.o.p("plane took off");
    }
    void fly(){
        S.o.pln("plane is flying");
    }
    void carryCargo(){
        S.o.pln("carrying goods");
    }
    void land(){
        S.o.pln("Plane has landed");
    }
}
```

```
class FighterPlane{
    void takeOff(){
        S.o.pln("Plane took off");
    }
    void fly(){
        S.o.pln("plane is flying");
    }
    void carryWeapons(){
        S.o.pln("carry weapons");
    }
    void land(){
        S.o.pln("Plane has landed");
    }
}
```

```
class Demo{
    p.s.v.m(){
        PassengerPlane p = new PassengerPlane();
        CargoPlane c = new CargoPlane();
        FighterPlane f = new FighterPlane();
        P.takeOff();
        P.fly();
        P.carryPassenger();
        P.land();

        c.takeOff();
        c.fly();
        c.carryCargo();
        c.land();

        f.takeOff();
        f.fly();
        f.carryWeapon();
        f.land();
    }
}
```

* (With Inheritance) :

UML Diagram : (Unified modelling language) :



```
class plane{
void takeOff(){
 S.o.pln("Plane took off");
}
void fly(){
  S.o.pln("Plane is flying");
}
void land(){
 S.o.pln("Plane has landed");
}
}
```

```
class Cargo extends Plane{
   void carryCargo(){
      S.o.pln("carrying goods");
   }
}
class Passenger extends Plane{
   void carryPassenger(){
      S.o.pln("carrying passengers");
   }
}
class Fighter extends Plane{
   void carryWeapons(){
      S.o.pln("carrying weapons");
   }
}
```

```
class Demo{
  P.S.V.m(){
    Passenger p = new Passenger();
    Cargo c = new Cargo();
    Fighter f = new Fighter();
    p.takeOff();
    p.fly();
    p.carryPassenger();
    p.land();
    f.takeOff();
    c.fly();
    c.carryCargo();
    c.land();
    f.takeoff();
    f.fly();
    f.carryWeapon();
    f.land();
  }
}
```

Types of Methods in Inheritance:

i) Inherited methods: The methods inherited from the parent & used as it is within the child.

ii) Specialized methods: The methods written exclusively within child class.

iii) Overridden methods: The methods inherited from the parent but used after changing the implementation.

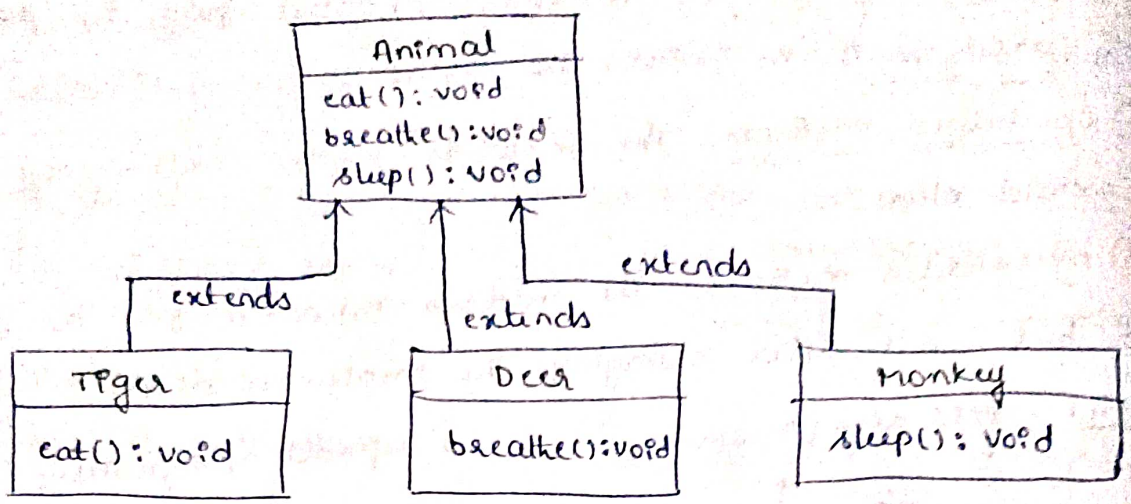
Note: void disp (Parameter_list) → Defination/Signature
{ } ≡≡ ↕ Implementation

- Eg for overridden methods: 13/2/23

```
class Animal {
void eat () {
  S.O.pln ("Animal is eating");
 }
 void breathe () {
   S.O.pln ("Animal is breathing");
 }
 void sleep () {
   S.O.pln ("Animal is sleeping");
 }
}

class Tiger extends Animal {
   void eat () {
     S.O.pln ("Tiger is hunting & eating");
   }
}

class Deer extends Animal {
  void breathe () {
    S.O.pln ("Deer is breathing gracefully");
  }
}

class Monkey extends Animal {
   void sleep () {
     S.O.pln ("monkey is sleeping on trees");
   }
}
```

```
class AnimalDemo {
  P.S.V.m () {
   Tiger t = new Tiger();
   Deer d = new Deer();
   Monkey m = new Monkey();
   t.eat(); Tiger is hunting & eating
   t.breathe(); AIB
   t.sleep(); AIS
   d.eat(); AIE
   d.breathe(); DIB
   d.sleep(); AIS
   m.eat(); AIE
   m.breathe(); AIB
   m.sleep(); MISOT
  }
}
```

- UML diagram:

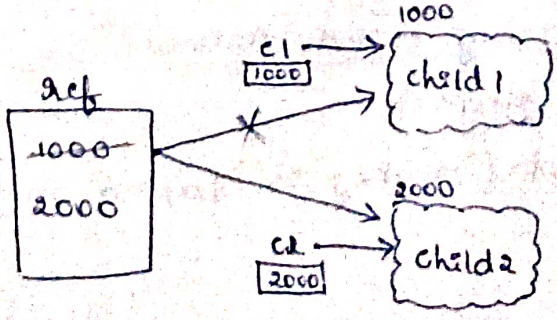


Note: A parent type reference can point to child class object & also call the inherited & overridden methods of child class. (parent type ref cannot call to specialized method of child class) directly
" can be done by downcasting

eg:

```
class parent{
 void disp1(){
   S.O.pln("Parent disp1");
 }
 void disp2(){
   S.O.pln("Parent disp2");
 }
}
class child1 extends Parent{
 void disp1(){
   S.O.pln("child1 disp1");
 }
}
class child2 extends Parent{
 void disp2(){
   S.O.pln("child2 disp1");
 }
}
```

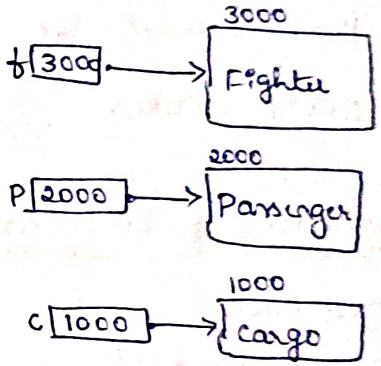
```
class Demo{
  P.S.V.m(){
    child1 c1 = new child1();
    child2 c2 = new child2();
    Parent ref;
    ref = c1;        // 1000
    (parent) ← (child)
    ref = (Parent) c1   // implicit
    ref.disp1(); // child1 disp1
    ref.disp2(); // Parent disp2
    ref = c2;        // 2000
    ref.disp1(); // Parent disp1
    ref.disp2(); // child2 disp2
  }
}
```



the above concept along with method overriding is used to implement True Runtime Polymorphism in Java.

eg:
```
class Plane{
  void land(){
    s.o.pln("Plane has landed");
  }
}

class Cargo extends Plane{
 void land(){
  s.o.pln("Landed with heavy goods");
 }
}

class Passenger extends Plane{
 void land(){
  s.o.pln("Landed with people");
 }
}

class Fighter extends Plane{
 void land(){
  s.o.pln("Landed with weapons");
 }
}
```

```
class Demo{
 P.s.v m(){
  Cargo c = new Cargo();
  Passenger p = new Passenger();
  Fighter f = new Fighter();
  Airport a = new Airport();
  a.allowPlane(c);
  a.allowPlane(f);
  a.allowPlane(p);
 }
}
```

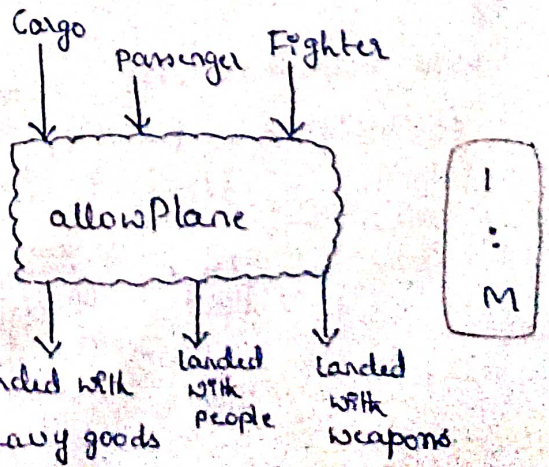



```
class Airport{                    (Cargo) 1000
 void allowPlane(Cargo ref){
   ref.land();
 }                                (Fighter) 3000
 void allowPlane(Fighter ref){
   ref.land();
 }                                (Passenger) 2000
 void allowPlane(Passenger ref){
   ref.land();
 }
}
```

Not best approach
False polymorphism

```
class Airport{                (cargo) 1000  (Passenger) 2000  (Fighter) 3000
 void allowPlane(Plane ref){
   ref.land();
 }
}
```

"Runtime Polymorphism"



- when a single method actually has the capability to perform multiple jobs based on the input we term it as Actual / True polymorphism. It is called as Runtime polymorphism because address of the object will be available to us only during execution.

Method overriding: The process of providing an alternate / new implementation for inherited methods of parent class within the child class is called as method overriding.

Note: Runtime polymorphism is also called as Late Binding / Dynamic binding / Dynamic method dispatch.

- polymorphism means multiple forms. It is a technique that allows the object / method to work from different perspectives

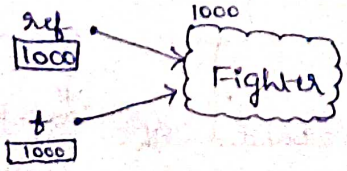
Upcasting & Downcasting:

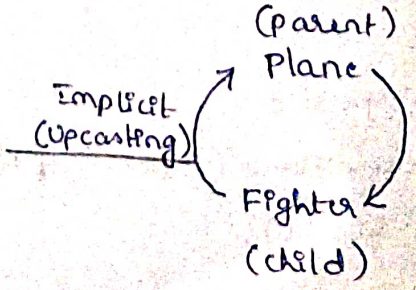
19/8/23

```
class Plane{
    void land(){
        S.o.pln("Plane has landed");
    }
}

class Fighter extends Plane{
    void land(){
        S.o.pln("Landed with weapons");
    }
    void launchMissile(){
        S.o.pln("Missile fired");
    }
}
```

specialized method

```
class Demo{
    P.S.V.m(){
        Fighter f = new Fighter();
        Plane ref;
        ref = (Plane) f;   // implicit
        ref.land();        // LNW
        ref.launchMissile(); // parent class
        ((Fighter)(ref)).launchMissile();
        // Missile fired
    }
}
```





- The process of converting child type variable to parent type is called Upcasting whereas the reverse is called Downcasting.
- Downcasting is used to invoke specialized methods of child class using variable of type parent.

Applications of final keyword:

i) A method marked as final can never be overridden.
ii) A class marked as final cannot be inherited.
iii) A variable marked as final will become a constant & its value cannot be changed.

eg: i) class Plane {
    final void land(){
        S.o.pln("Landing at 200 kmph");
    }
}
class Passenger extends Plane {
    void land(){
        S.o.pln("Landing at 300 kmph");
    }
}
↕ Error

ii) class Plane {
    ≡
}
class Passenger extends Plane{
    ≡
}
class Fighter extends Passenger{
    ≡
}
final class Supersonic extends Fighter{
    ≡
}
class Cargo extends Supersonic{
    ≡
}
↕ Error

iii) final float PI = 3.14f;
PI = 46.66f; // Error

Access Specifiers/Modifiers:

Public → accessed anywhere
Private → within same class
default → same package
Protected → same package & diff pkg with related class

```
class A{
  Public a;
  (default; b;
  Protected c;
  Private d;
  void disp(){
    a✓ b✓
    c✓ d✓
  }
}
```

```
class B extends A{
  void disp1(){
    a✓ b✓
    c✓ d✗
  }
}
```

```
class C{
  void disp2(){
    a✓ b✓
    c✓ d✗
  }
}
```

Package1

```
class D extends A{
  void disp3(){
    a✓ b✗
    c✓ d✗
  }
}
```

```
class E{
  void disp4(){
    a✓ b✗
    c✗ d✗
  }
}
```

Package2

	Public / ~~Private~~ (a)	Protected (c)	default (b)	Private (d)
within same class	✓	✓	✓	✓
within related class within same package	✓	✓	✓	✗
within unrelated class same package	✓	✓	✓	✗
within related class different package	✓	✓	✗	✗
within unrelated class diff package	✓	✗	✗	✗

Note: when method overriding takes place the child method cannot have an access specifier weaker than the parent method.

(Security) (Terminology)

1. Private → weakest
2. default → weaker
3. Protected → weaker
4. Public → weak

Parent	child
Public	Public
Protected	protected, Public
default	default, protected, Public
Private	no inheritance takes place

Private variables & methods are not inherited into child.

```
class A{
    void disp1(){
    ==
    }
    void disp2(){
    ==
    }
    private void disp3(){
    ==
    }
}
```

```
class B extends A{
    void disp1(){
    ==
    }                    } overridden method
    void disp3(){
    ==
    }                    } specialized method
    void disp2(){
    ==
    }                    } inherited method
}
```

Rules for Method overriding:

21/8/23

- Methods marked as final & private cannot be overridden.
- During overriding the signature of the parent class method should not be changed unless it is a co-variant return type.
- Access specifier of the child class method should not be weaker than parent class method.

eg: i)
```
class A{
    void disp(){
        S.O.pln("Inside A's disp");
    }
}

class B extends A{
    int disp(){                    NO-overriding
        int a=10, b=20;            Error
        return (a+b);
    }
}
```

```
void disp(){
    S.O.pln("Inside A's disp");
}
```

ii)
```
class A{
    void disp(){
        S.O.pln("Inside A's disp");
    }
}

class B extends A{
    int disp(int a, int b){
        return(a+b);                (overloaded)
    }                               NO overriding
                                    No Error
    void disp(){
    ==
    }                               → Inherited
}
```

iii)
```
class A{
    void disp(){
        S.O.pln("A's disp");
    }
}
```

```
class B extends A{
    int disp(int a, int b){    } (overloaded)
        return (a+b);
    }
    void disp(){               } overriding
        S.O.pln("B's disp");   } No Error
    }
}
```

Co-variant return type:

The types that have a parent-child relationship are called co-variant types.

```
class Animal{
    ==
}
class Dog extends Animal{
    ===
}
```

```
class Parent{
    Animal disp(){
        ==
        Animal a = new Animal();
        ==
        return a;
    }
}
class Child extends Parent{
    Dog disp(){
        ==
        Dog d = new Dog();
        ==
        return d;
    }
}
```

(is a)

overriding takes place as Dog & Animal are co-variant types.

Q Explain the diamond shaped problem.

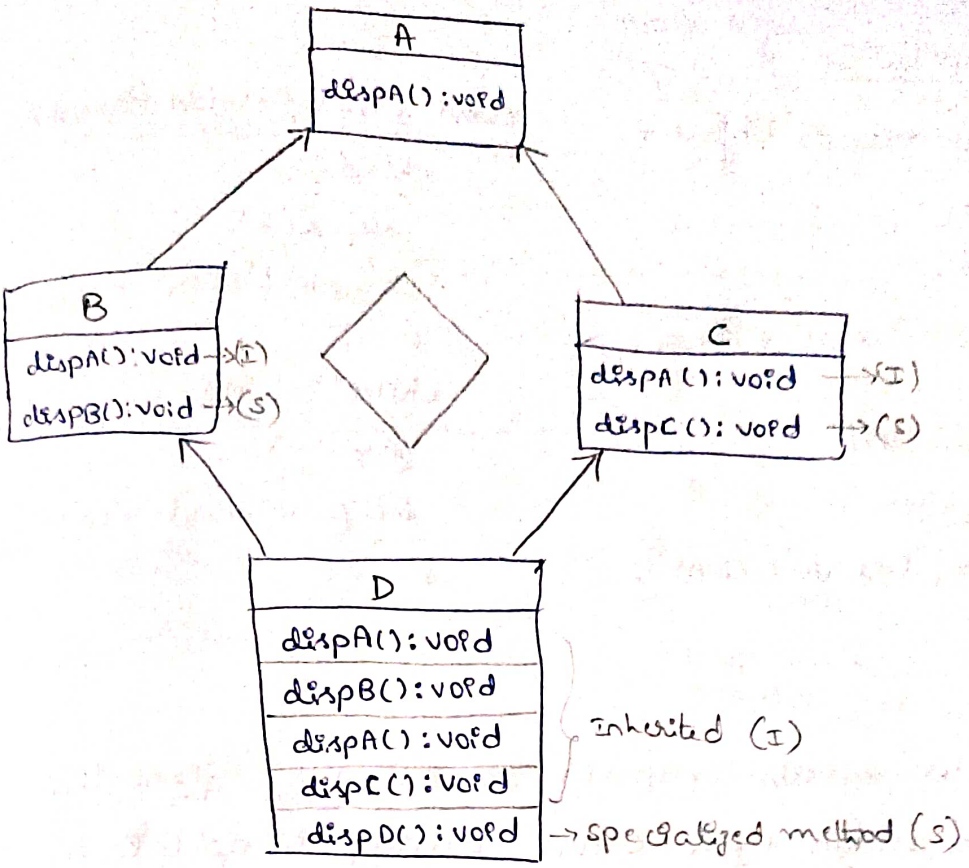
```
class A{
    void dispA(){
        ==
    }
}
class B extends A{
    void dispA(){
        ==
    }
    void dispB(){
        ==
    }
}
class C extends A{
    void dispA(){
        ==
    }
    void dispC(){
        ==
    }
}
```

(inherited)

```
class D extends B,C{
    void dispA(){
        ==
    }
    void dispB(){
        ==
    }
    void dispA(){
        ==
    }
    void dispC();
        ==
    }
    void dispD(){
        ==
    }
}
```

```
main(){
    D d = new D
    d.dispA()
}
```

Ambiguity ERROR



. If multiple inheritance is allowed, it may result in the formation of a diamond-shaped UML. This means that 2 methods with the same signature may get inherited into the same class leading to ambiguity when a call to such methods is made. This is called as Diamond-shaped pblm & to avoid it multiple inheritance is not allowed in Java.

④ Why is the parent constructor executed before child constructor?

→ (Before compilation)

```
class Parent{
    Parent(){
        S.o.pln("Parent o cons");
    }
    Parent(int a){
        S.o.pln("Parent 1 cons");
    }
}
```

```
class Child extends Parent{
    child(){
        S.o.pln("child o cons");
    }
    child(int a){
        S.o.pln("child 1 cons");
    }
}
```

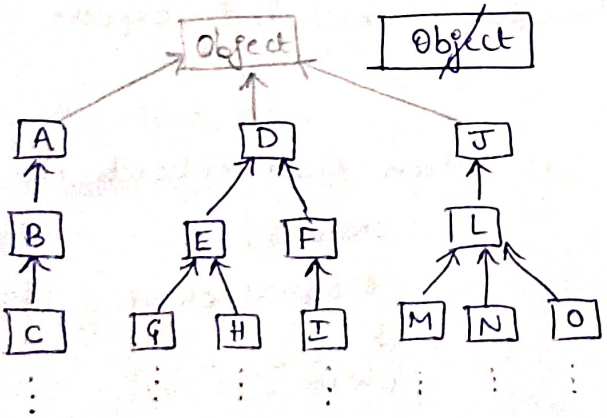
After compilation ⇒

```
class Parent extends Object {
    Parent () {
        super ();
        S.O.pln ("Parent 0 cons");
    }
    Parent (int a) {
        super ();
        S.O.pln ("Parent 1 cons");
    }
}

class Child extends Parent {
    child () {
        super ();
        S.O.pln ("child 0 cons");
    }
    child (int a) {
        super ();
        S.O.pln ("child 1 cons");
    }
}
```

- The compiler inserts super() as the very first line of each constructor within each class during compilation.
- Hence, constructor of the parent most class will be executed first & so on until child class constructor.

Q Why Object class is called as Root node / Master node?

- Every parent-child hierarchy will have the object class at top most level. This means that very parent-child relationship originates from Object class. Hence it is called as root / master node.



Constructor chaining: 22/8/23

- The process of calling one constructor from within another constructor is called as constructor chaining.
- If the chaining takes place within same class then it is called local chaining. (this can be achieved by using this)

```
class Parent {
  Parent() {
    S.O.pln("Inside Parent constructor");
  }
}
class child extends Parent {
  child() {
    super();
    S.O.pln("Inside child constructor");
  }
}
child c = new child();
```

o/p → ① ② ③

"constructor chaining b/w classes"

```
class Demo {
  Demo() {
    this(10);
    S.O.pln("Inside 0 para cons");
  }
  Demo(int a) {
    this(10,20);
    S.O.pln("Inside 1 para cons");
  }
  Demo(int a, int b);
    S.O.pln("Inside 2 para cons");
  }
}
Demo d = new Demo();
```

o/p → ③ ② ①

"constructor chaining within same class"

② can super & this be used simultaneously?

- Yes. As shown below both the keywords can be used simultaneously.
- super is inserted by the compiler in order to execute parent constructor. whereas this is used to perform local chaining.

```
class Parent {
  Parent() {
    S.O.pln("Inside Parent cons");
  }
}
class Demo extends Parent {
  Demo() {
    super();
    this(10);
    S.O.pln("Inside 0 para cons");
  }
  Demo(int a) {
    super(); (Executed once)
    this(10,20);
    S.O.pln("Inside 1 para cons");
  }
  Demo(int a, int b);
    super();
    S.O.pln("Inside 2 para cons");
  }
}
Demo d = new Demo();
```

(i) (ii) (iii) (iv) (v) (vi)

① ④ ③ ②

one constructor for one object will be executed only once.

```
class Parent{
  Parent(){
   S.o.pln("Inside Parent cons");   (4)
  }
  static{
   S.o.pln("Inside static block
                Parent");   (1)
  }
  {
   S.o.pln("Inside non-static
              block parent");   (3)
  }
}
```

Steps: 1. Static block → Parent
2. Static block - child
3. Non-static block → parent
4. constructor → Parent
5. Non-static block - child
6. constructor - child

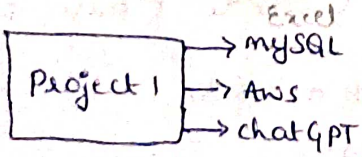
```
class child extends Parent{
  Child(){
   S.o.pln("Inside child cons");   (6)
  }
  static{                          (2)
   S.o.pln("Inside Static block child");
  }
  {
  S.o.pln("Inside non-static
              block child");   (5)
  }
}

class Demo{
    P.s.v.m(){
     child c=new child();
}  }
```

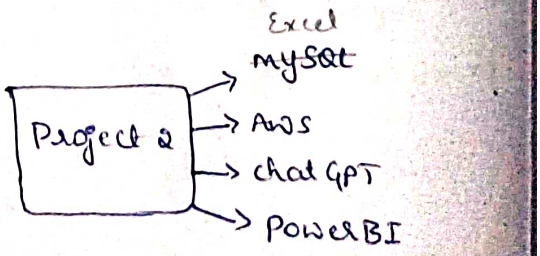
Disadvantages of Inheritance:

- The only disadvantage is that any changes made to Parent class will reflect in the child class.

eg:
```
class Project1{
  void connection1(){
  S.o.pln("connect to mySQL");   (Excel)
  }
  void connection2(){
   S.o.pln("connect to AWS");
  }
  void connection3(){
    S.o.pln("connect to chatGPT");
  }
}
```

```
class Project2 extends Project1{
  void connection4(){
    S.o.pln("connect to POWERBI");
  }
}
```





- In the above eg. PowerBI is the only specialized requirement for Project2. Hence implementation of mySQL, AWS, chatGPT has been inherited from project1. Later on project1 has changed mySQL to Excel. This change even though unwanted also happens in project2.

Delegation Model:

- Whenever an is-a relationship is not possible b/w 2 classes, we can establish has-a relationship using delegation model.
- Delegation model supports 2 types of objects:
  i) composed objects: It gets destroyed along with the distruction of main object. It must be created as an instance variable within the main object.
  ii) Aggregate object: It doesnot get destroyed even after distruction of main object. It must be passed as a parameter to one of the methods of class of the main object.

Note:

```
class Dog{          is-a          class Animal{
==                                ==
}                                 }
```

Dog extends Animal ✓ (is-a)

```
class Car{          has-a          class Radio{
==                                 ==
}                                  }
```

Car extends Radio ✗

- is-a relation:

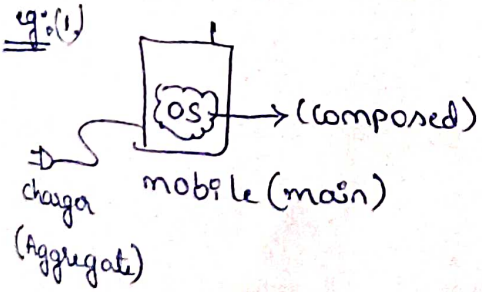
child ⟶ Parent

- has-a relation:

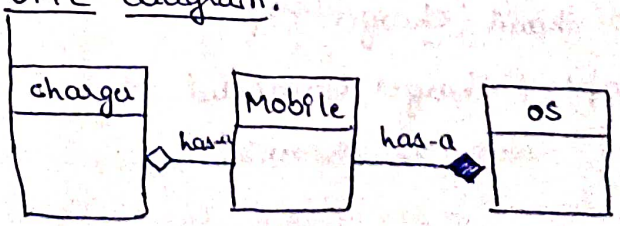
main object ◇⟶ Aggregate object

main ◆⟶ composed

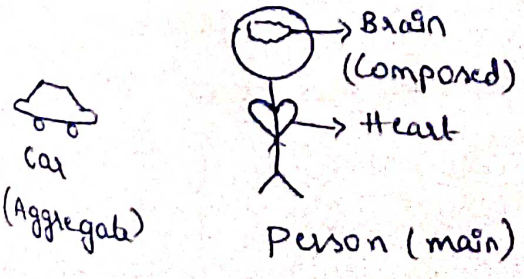
eg:(1)

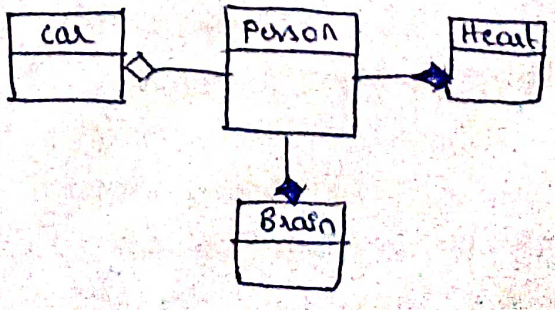


UML diagram:



(2)



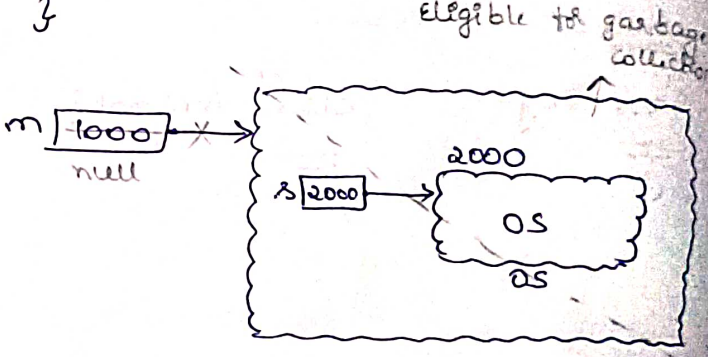


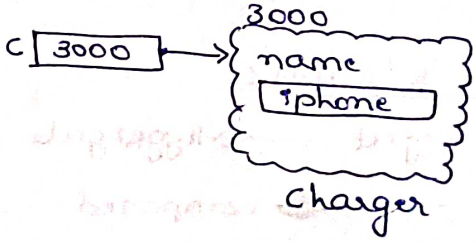
Eg:

28/8/23

```
class OS{
    OS(){                    (1)
     s.o.pln ("os Installed");
    }
  void checkOS(){            (5)
    s.o.pln("os is working fine");
  }
}

class Charger{
  String name;
  Charger (String n){        (2)
    s.o.pln("charger ready for
                    charging");
   name=n;
  }
  void checkCharger(){       (6)
    s.o.pln("charger is working
                    fine");
  }
}

class Mobile{
  OS s=new OS();
  Mobile(){
   s.o.pln("mobile created with
                    OS installed");   (3)
  }
  void hasA(Charger c){      (4)
    s.o.pln("Charger connected to
                    mobile phone");
  }
}
```

```
class DelegationDemo{
    P.S.V.m(){
      Mobile m=new Mobile();
      Charger c=new Charger("iphone");
      m.hasA(c);
      m.s.checkOS();
      c.checkCharger();
      m=null; (to destroy mobile obj)
      c.checkCharger();
      m.s.checkOS(); → Error
    }
}
```





Abstract Method :

A method without implementation is incomplete & must be marked as abstract.

Abstract class:

A class which contains abstract methods is an incomplete class & must be marked as abstract.

Note: • During inheritance if none of the child classes are using implementation of parent class methods / if every child gives diff implementation for its methods, then parent class methods must be marked abstract.

• It is the responsibility of the child class to provide implementation for all abstract methods of the parent class. If the child fails to do so, even the child class must be marked abstract.

• objects of Abstract classes: cannot be created.

eg: 1.

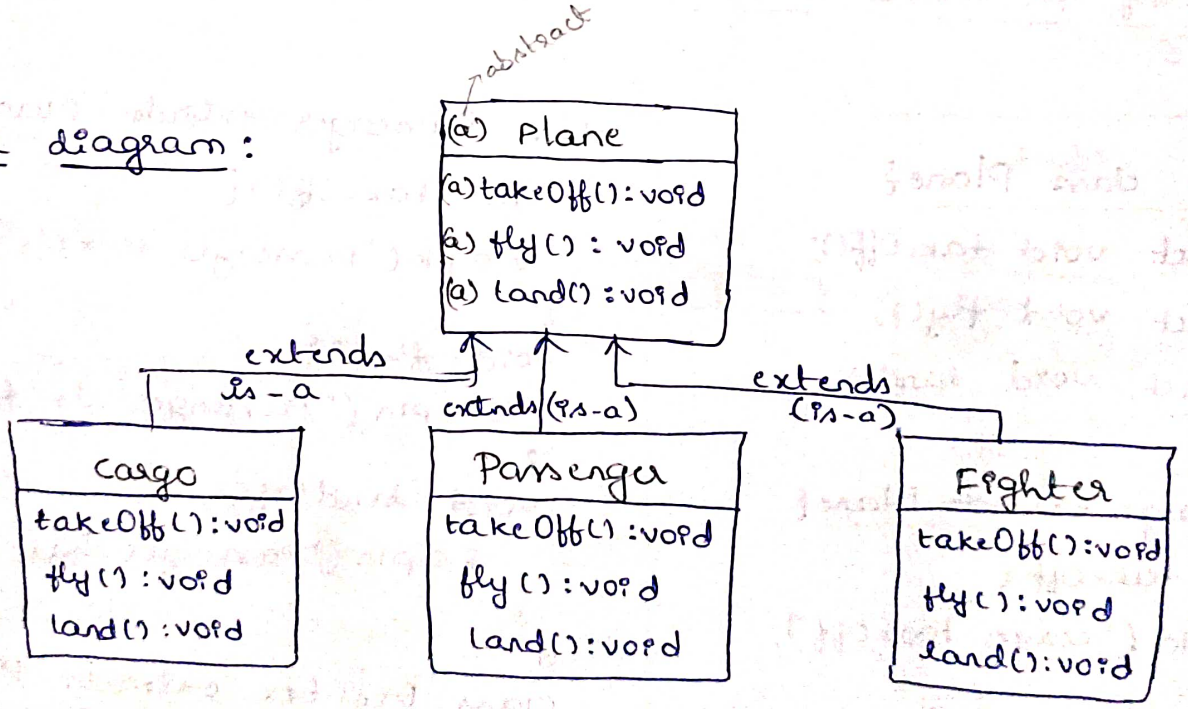
```
abstract class Plane{
  abstract void takeOff();
  abstract void fly();
  abstract void land();
}
class Cargo extends Plane{
  void takeOff(){
    s.o.pln("cargo tookoff");
  }
  void fly(){
    s.o.pln("cargo is flying");
  }
  void land(){
    s.o.pln("cargo has landed");
  }
}
```

pure abstract class

```
class Passenger extends Plane{
  void takeOff(){
    s.o.pln("Passenger tookoff");
  }
  void fly(){
    s.o.pln("Passenger is flying");
  }
  void land(){
    s.o.pln("Passenger has landed");
  }
}
class Fighter extends Plane{
  void takeOff(){
    s.o.pln("Fighter tookOff");
  }
  void fly(){
    s.o.pln("Fighter is flying");
  }
  void land(){
    s.o.pln("Fighter has landed");
  }
}
```

```
class Airport {
    void allowPlane (Plane ref) {
        ref.takeOff();
        ref.fly();
        ref.land();
    }
}
class Demo{
    P.S.V.m(){
    Cargo c = new Cargo();
    Passenger p = new Passenger();
    Fighter f = new Fighter();
    Airport a = new Airport();
    a.allowPlane(c);
    a.allowPlane(p);
    a.allowPlane(f);
    }
}
```

UML diagram:



q:2 :

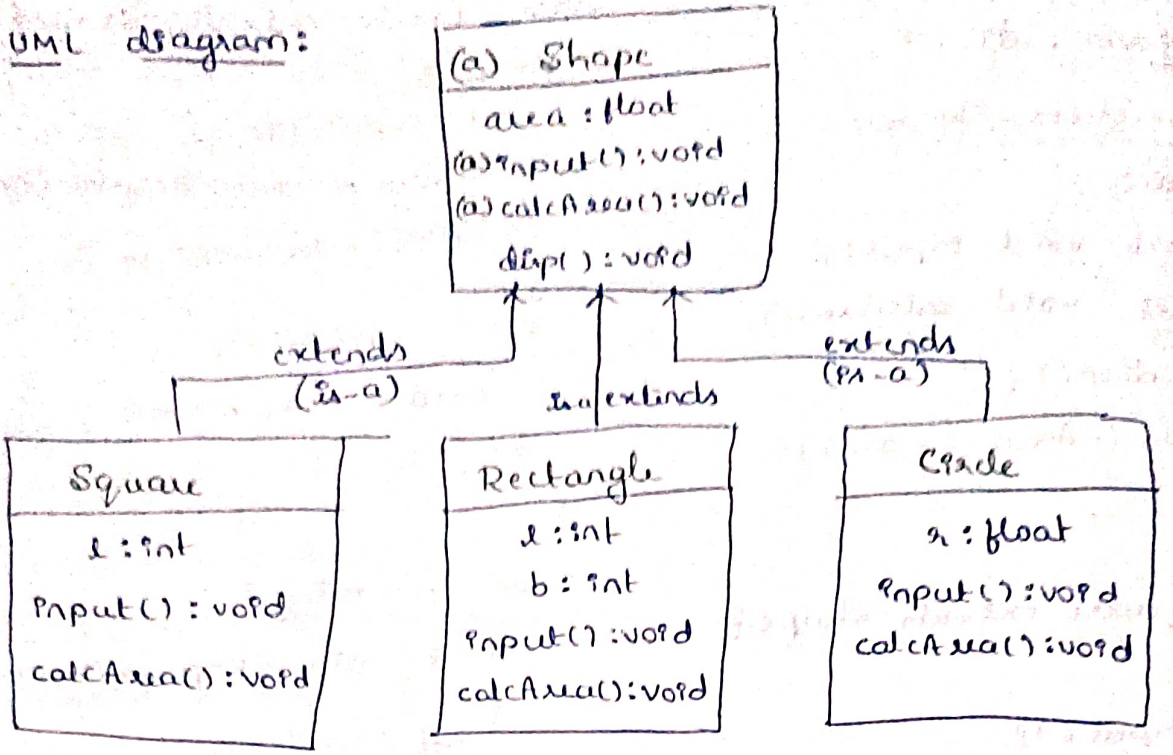
```
import java.util.*;
abstract class Shape{
float area;
abstract void input();
abstract void calcArea();
void disp(){
 S.o.pln("Area="+area);
}
}
class Square extends Shape{
int l;
void input(){
 Scanner sc=new Scanner(System.in);
 S.o.pln("Enter length of square");
  l=sc.nextInt();
}
void calcArea(){
   area=l*l;
}
}
class Rectangle extends Shape{
int l, b;
void input(){
 Scanner sc=new Scanner(System.in);
 S.o.pln("Enter length:");
  l=sc.nextInt();
 S.o.pln("Enter breadth:");
  b=sc.nextInt();
}
void calcArea(){
  area=l*b;
}
}
```

Implements abstract class

```
class Circle extends Shape{
float r;
void input(){
 Scanner sc=new Scanner(System.in);
 S.o.pln("Enter radius:");
}
void calcArea(){
   area=3.14f*r*r;
}
}
class Geometry{
   void allowShapes(Shape ref)
   {
     ref.input();
     ref.calcArea();
     ref.disp();
   }
}
class Demo{
  P.S.V.M(){
   Square s=new Square();
   Rectangle r=new Rectangle();
   Circle c=new Circle();
   Geometry g=new Geometry();
   g.allowShapes(s);
   g.allowShapes(r);
   g.allowShapes(c);
  }
}
```

s r c

UML diagram:

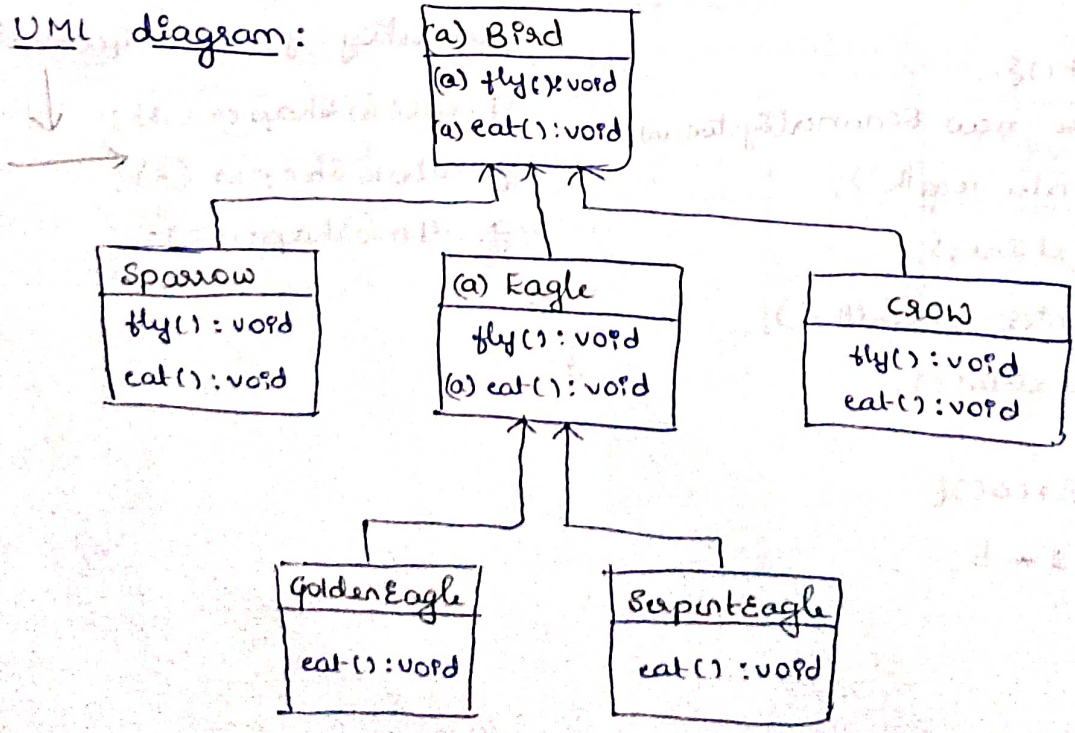


Note: 29/8/23

- A method with implementation is called a concreate method.
- An abstract class which contains only abstract methods is called a pure abstract class.
- An abstract class which contains both abstract methods & concreate methods is called impure abstract class.

eg: 3 =>

UML diagram:



```
abstract class Bird{
  abstract void fly();
  abstract void eat();
}
class Sparrow extends Bird{
  void fly(){
    S.O.pln("Flying at low altitude");   ①
  }
  void eat(){
    S.O.pln("Eating insects & grains");   ②
  }
}
class
abstract class Eagle extends Bird{
  void fly(){
    S.O.pln("Flying at high altitude");   ⑤ ⑦
  }
  abstract void eat();
}
class Crow extends Bird{
  void fly(){
    S.O.pln("Flying at medium altitude");   ③
  }
  void eat(){
    S.O.pln("Eating anything available");   ④
  }
}
class GoldenEagle extends Eagle{
  void eat(){
    S.O.pln("Eating seafood");   ⑥
  }
}
class SuperEagle extends Eagle{
  void eat(){
    S.O.pln("Eating snakes");   ⑧
  }
}
class Sky{
  void allowBird(Bird ref)
  {
    ref.fly();
    ref.eat();
  }
}
class Demo{
  P S V m(){
  Sparrow s=new Sparrow();
  Crow c=new Crow();
  GoldenEagle g=new GoldenEagle();
  SuperEagle sp=new SuperEagle();
  Sky sk=new Sky();
  sk.allowBird(s);
  sk.allowBird(c);
  sk.allowBird(g);
  sk.allowBird(sp);
  }
}
```

- Address of child object can be collected by parent class & grandparent class also.

Note:

- If the child fails to provide implementation for all the methods of parent class, it can simply be marked abstract without re-declaring abstract methods of parent within child class.
- A class with only concrete methods can be marked abstract to ensure no object of it is created.

eg: i)

```
abstract class Mathematics {

    abstract void add();
    abstract void sub();
    abstract void mul();
    abstract void div();
}

abstract class Addition extends Mathematics {
    void add(){
      ==
    }
    abstract void sub();
    abstract void mul();
    abstract void div();
}
```

⟺ (equivalent class)

```
abstract class Addition
        extends Mathematics{
    void add(){
      ==
    }
}
```

```
Mathematics m = new Mathematics();
Addition a = new Addition();
```

} Error

↓

cannot create object of abstract class.

eg: ii)

```
abstract class Demo{
    void add(){
      ==
    }
    void sub(){
      ==
    }
    void mul(){
      ==
    }
    void div(){
      ==
    }
}
```

```
Demo d = new Demo();   // Error
```

⇒ As per Java community any class that is not representing a real world object should not be instantiated hence must be marked as abstract. (cannot create object of class)

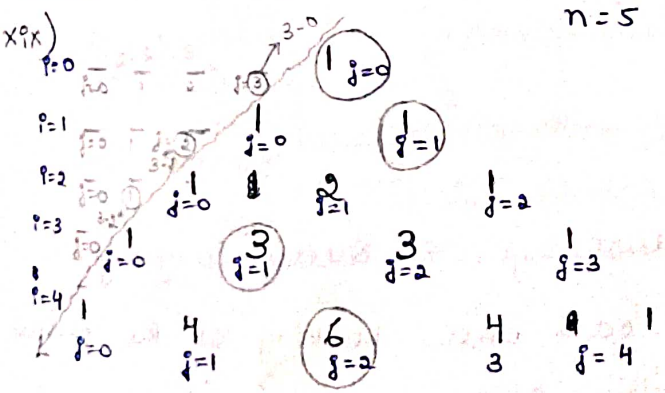
eg: class Animal must be marked abstract even though it is completly implemented, as in the real world there is no object named Animal, it is a name given to represent a group of objects.

(a)
```
abstract class Animal{
    ==
}
class Tiger extends Animal{
    ==
}
```

(b)
```
abstract class Plane{
    ==
}
class Cargo extends Plane{
    ==
}
```

(c)
```
abstract class Bird{
    ==
}
abstract clan extends Bird{
    ==
}
class GoldenEagle extends Eagle{
    ==
}
```

Pattern Matching :



$${}^{i}C_{j} = \frac{i!}{(i-j)!\,(j)!}$$

$${}^{0}C_{0} = \frac{0!}{0! \times 0!} = 1$$

$${}^{1}C_{1} = \frac{1!}{(1-1)! \times 1!} = 1$$

$${}^{3}C_{1} = \frac{3!}{2! \times 1!} = 3$$

$${}^{4}C_{2} = \frac{4 \times 3 \times 2!}{(4-2)! \times 2 \times 1} = 6$$

```
class PascalTriangle{
  psvm(){
    int i, j, x;
    for(i=0; i<=4; i++){
      for(j=0; j<=3-i; j++){
        s.o.p(" ");
      }
      for(j=0; j<=i; j++){
        x = fact(i) / (fact(i-j) * fact(j));
        s.o.p(x+" ");
      }
      s.o.pln();
    }
  }
  public static int fact(int a){
    int i, prod=1;
    for(i=1; i<=a; i++){
      prod = prod * i;
    }
    return prod;
  }
}
```

```
import java.util.*;
class PascalTriangle{
  psvm(){
    Scanner sc=new Scanner(System.in);
    s.o.p("Enter the value");
    int n=sc.nextInt();
    int i, j, x;
    for(i=0; i<=n-1; i++){
      for(j=0; j<=(n-i)-2; j++){
```

for n values → psvm

30/8/23

Interfaces :

Introduction :

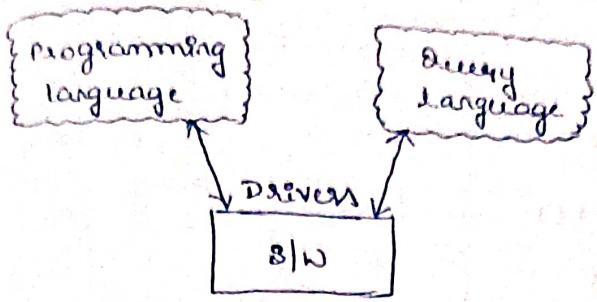
- Until 1990's programming languages & Query languages couldn't communicate which each other. Looking at the future advancements DBMS companies & Sun microsystems wanted to create a technology using which the above 2 could communicate.
- Hence, drivers were created which acted as an intermediary for programming languages & query languages.
- This led to standardization issues as
  *) Developers had to study different set of methods to work with different DBMS.

*) successfully working project couldn't be migrated to another DBMS without making changes to the existing project.

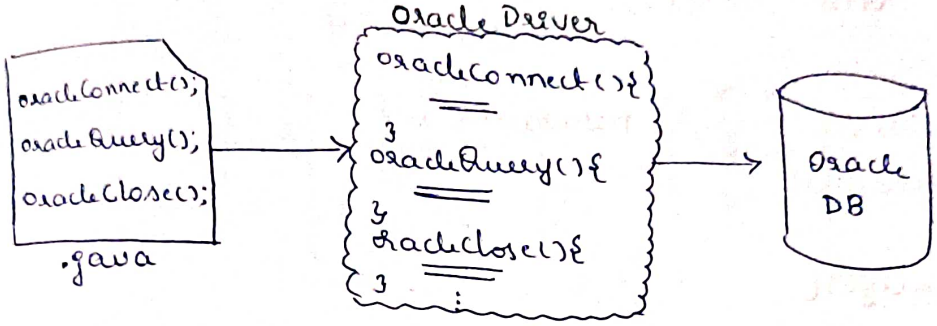
1990's

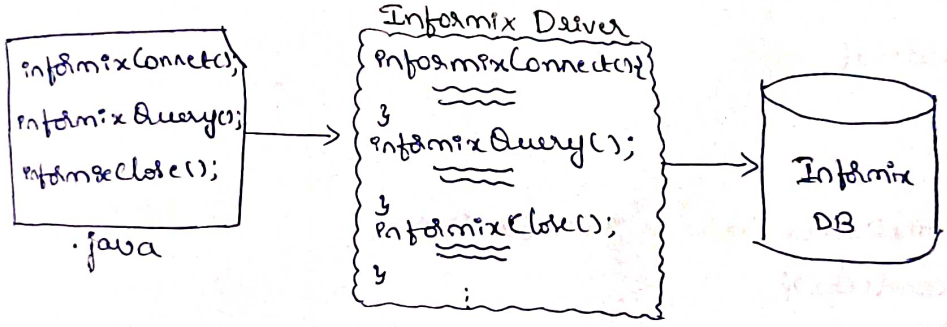
Oracle ⟶ Sun Microsystems

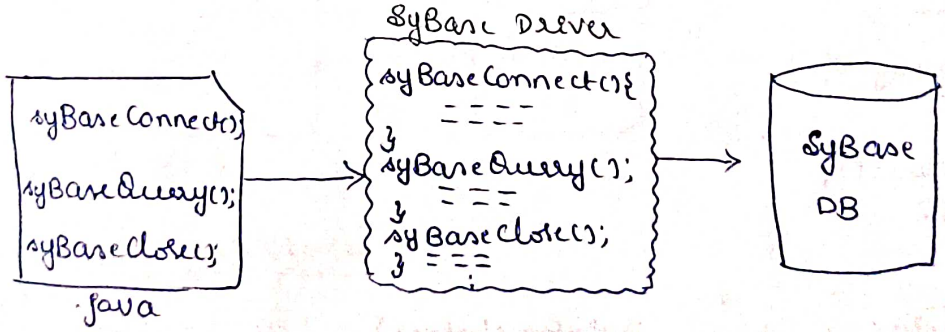
=>



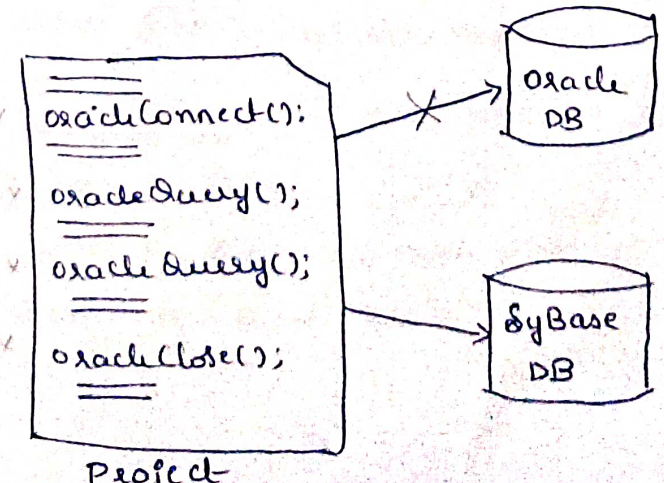
=>







=>

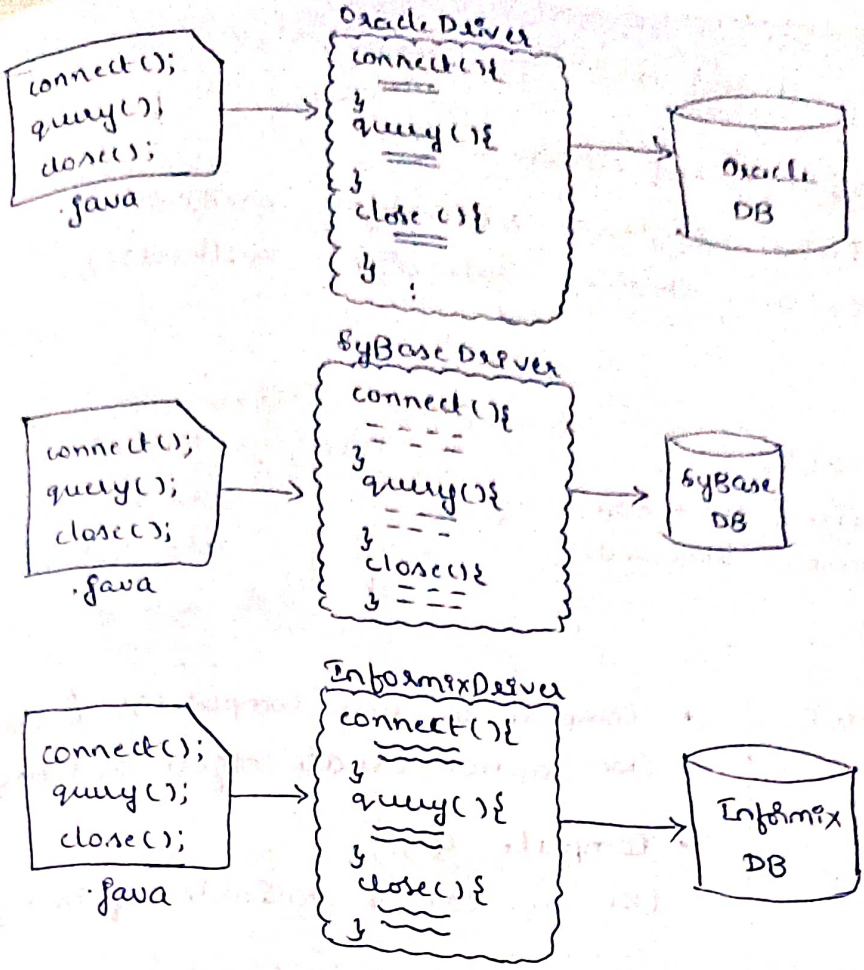


- In order to introduce standardization Java introduced the concept of interfaces.

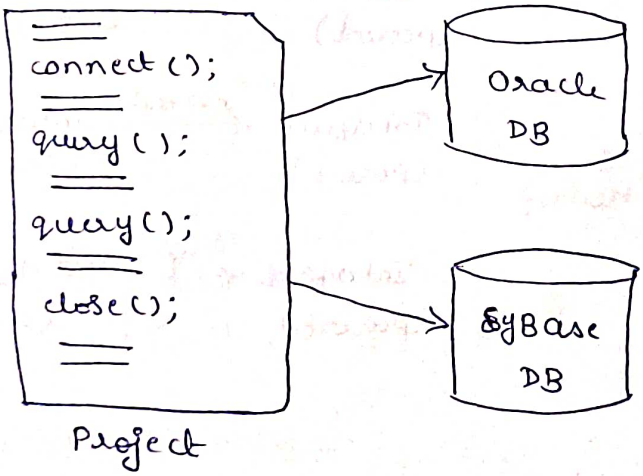
*** An Interface is a template for a class (blueprint). It defines a set of protocols that the class which implements the interface must follow.

=> interface DataBaseDriver{
    abstract void connect();
    abstract void query();
    abstract void close();
}
class OracleDriver extends DataBaseDriver{
    void connect(){
        ====
    }
    void query(){
        ====
    }
    void close(){
        ====
    }
}
class InformixDriver extends DataBaseDriver{
    void connect(){
        ======
    }
    void query(){
        ======
    }
    void close(){
        ======
    }
}
class SyBaseDriver extends DataBaseDriver{
    void connect(){
        =====
    }
    void query(){
        =====
    }
    void close(){
        =====
    }
}

=>



=>



Need for Interfaces :

- To Bring in standardization.
- To achieve 100% abstraction.

*** Note: Abstraction is the process of hiding the implementation details of a class.

* The parent class will not & need not know the implement-ation of child class.

Rules for using Interfaces:

1) Syntax:
```
interface InterfaceName {
    public abstract returntype method1();
    public abstract returntype method2();
        :
}
```

Eg:
```
interface Maths{
    public abstract void add();
    public abstract void sub();
}
```
<=>
```
interface Maths{
    void add();
    void sub();
}
```
→ By default public abstract will be there

ii)
```
interface compute {
    ======
}
```

- Compute cp = new Compute(); // Error (We cannot create object of interfaces)
- Compute cp; (We can create variables of interface)

iii)
```
interface Maths{
    void add();
    void sub();
}

class Calculation extends Maths{
    ======
}
```
×

class ← extends class
(parent) (child)

Interface ← extends Interface
(parent) (child)

Interface ← implements class
(parent) (child)

iv)
```
interface maths{
    void add();
    void sub();
}
```
- ```
  class Calculation implements Maths{
      void add(){
          ======
      }
  }
  ```
  ×
  (The child class should provide implementation to all the abstract methods of parent class. If child class fails to do than than child class must be marked as abstract)
- ```
  abstract class Calculation implements Maths{
      public void add(){
          ======
      }
  }
  ```
  ✓

```
class Calculation implements Maths{
        void add(){
        ==========
        }
        void sub(){
        ==========
        }
}
```

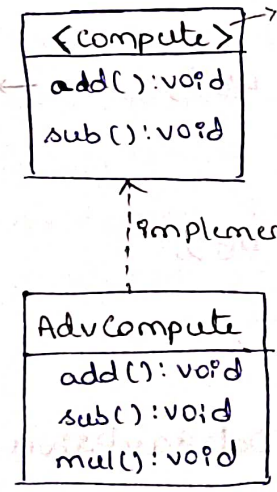
(The access specifiers of child class method cannot be weaker than access specifier of parent class method)

```
class Calculation implements Maths{
    public void add(){
    ==========
    }
    public void sub(){
    ==========
    }
}
```

Implementation of Interfaces:

31/8/23

eg: 1.

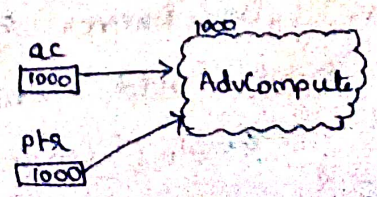



```
interface Compute{
    void add();
    void sub();
}

class AdvCompute implements Compute{
    Public void add(){
        int a=10, b=20;
        int c=a+b;
        S.o.pln(c);
    }
    Public void sub(){
        int a=10, b=20;
        int c=a-b;
        S.o.pln(c);
    }
    void mul(){
        int a=10, b=20;
        int c=a*b;
        S.o.pln(c);
    }
}

class Demo{
    P.S.V.m(){
        AdvCompute ac=new AdvCompute();
        ac.add(); // 30
        ac.sub(); // -10
        ac.mul(); // 200
        Compute ptr;
        ptr=(Compute)ac; // Upcasting (implicit type casting)
        ptr.add(); // 30
        ptr.sub(); // -10
        ptr.mul(); // Error (parent type ref cannot call specialized method of child class)
        ((AdvCompute)(ptr)).mul(); // 200
    }
}
```

(public abstract) by default

specialized method

Downcasting (explicit)



Implementing Runtime polymorphism using Interfaces:

Runtime Polymorphism can be achieved by using method overriding & parent type variable (class / interface) holding address of child class object.

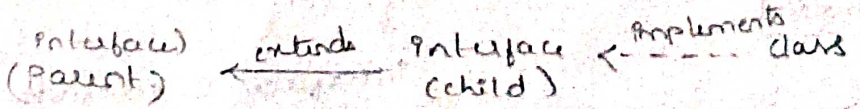
eg:

```
Interface DBDataBaseDriver{
    void connect();
    void query();
    void close();
}

class OracleDriver implements DataBaseDriver{
    Pubic void connect(){
        S.o.pln("connect to Oracle DB");
    }
    Public void query(){
        S.o.pln("query the oracle DB");
    }
    Public void close(){
        S.o.pln("close the oracle DB");
    }
}

class InformixDriver implements DataBaseDriver{
    Public void connect(){
        S.o.pln("Connect to Informix DB");
    }
    Public void query(){
        S.o.pln("query the Informix DB");
    }
    Public void close(){
        S.o.pln("close the Informix DB");
    }
}

class SyBaseDriver implements DataBaseDriver{
    Public void connect(){
        S.o.pln("connect to SyBase DB");
    }
```

```
    public void query(){
        S.O.Pln(" Query the SyBase DB");
    }
    public void close(){
        S.O.Pln(" close the SyBase DB");
    }
}

class Connection{
    void allowConnection(DataBaseDriver ref){
        ref.connect();
        ref.query();
        ref.close();
    }
}

class Demo{
    P.S.V m(){
        OracleDriver o = new OracleDriver();
        InformixDriver i = new InformixDriver();
        SyBaseDriver s = new SyBaseDriver();
        Connection c = new Connection();
        c.allowConnection(o);
        c.allowConnection(i);
        c.allowConnection(s);
    }
}
```

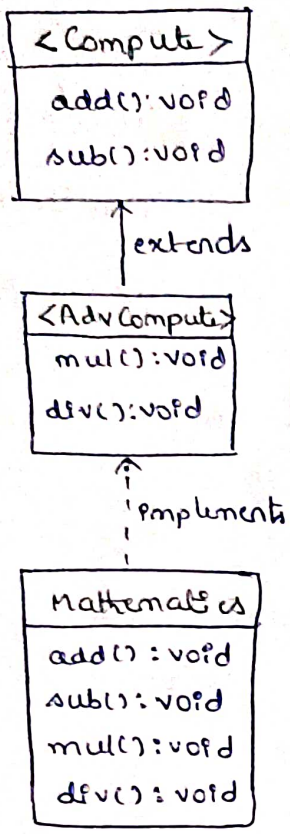
Note:

i) If a class implements an interface which already extends another interface, it is the responsibility of the child class to provide implementation for all the methods in the hierarchy. Otherwise the class must be marked abstract.



eg: P.T.O

eg:

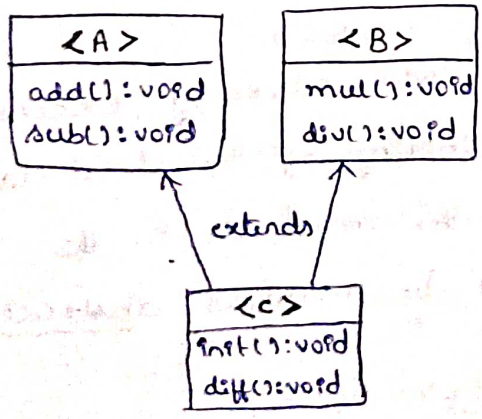



```
interface Compute{
    void add();
    void sub();
}
interface AdvCompute extends Compute{
    void mul();
    void div();
}
class Mathematics implements AdvCompute{
    public void add(){
        ==
    }
    public void sub(){
        ==
    }
    public void mul(){
        ==
    }
    public void subdiv(){
        ==
    }
}
```

ii) Multiple inheritance in Java can be achieved through interfaces.

iii) There will be no diamond-shaped pblm in this case.

eg: 1

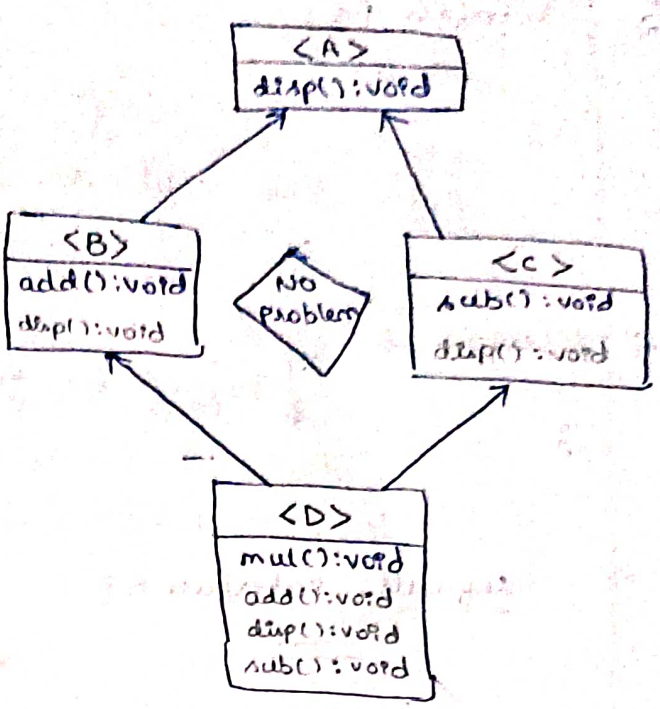



```
interface A{
    void add();
    void sub();
}
interface B{
    void mul();
    void div();
}
interface C extends A,B{
    void fnt();
    void diff();
}
```

```
void add();
void sub();
void mul();
void diff();
```

eg: 2

```
interface A{
  void disp();
}
interface B extends A{
   void add();
   void disp();
}
interface C extends A{
    void sub();
    void disp();
}
interface D extends B, C {
     void mul();
     void add();
     void disp();
     void sub();
}
```

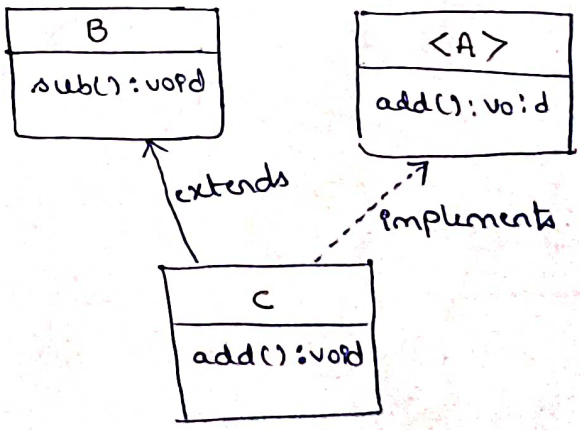


(Q) can extends & implements keywords be used simultaneously?

↳ Yes. But extends must come before implements.

eg:

```
interface A {
 void add();
}
class B{
 void sub(){
  int a=10,b=20,c;
   c=a-b;
   s.o.p(c);
 }
}
class C extends B implements A {
   public void add(){
        int a=10,b=20,c;
        c=a+b;
        s.o.pln(c);
   }
}
```



Note:

=> interface A{

int a;

}

Public final static int a;

=> interface B{

}

Tagged (or) Marker Interface

=> default interface A{

==

}

Public interface B{

==

}

Public Protected interface C{

==

}

Private interface D{

==

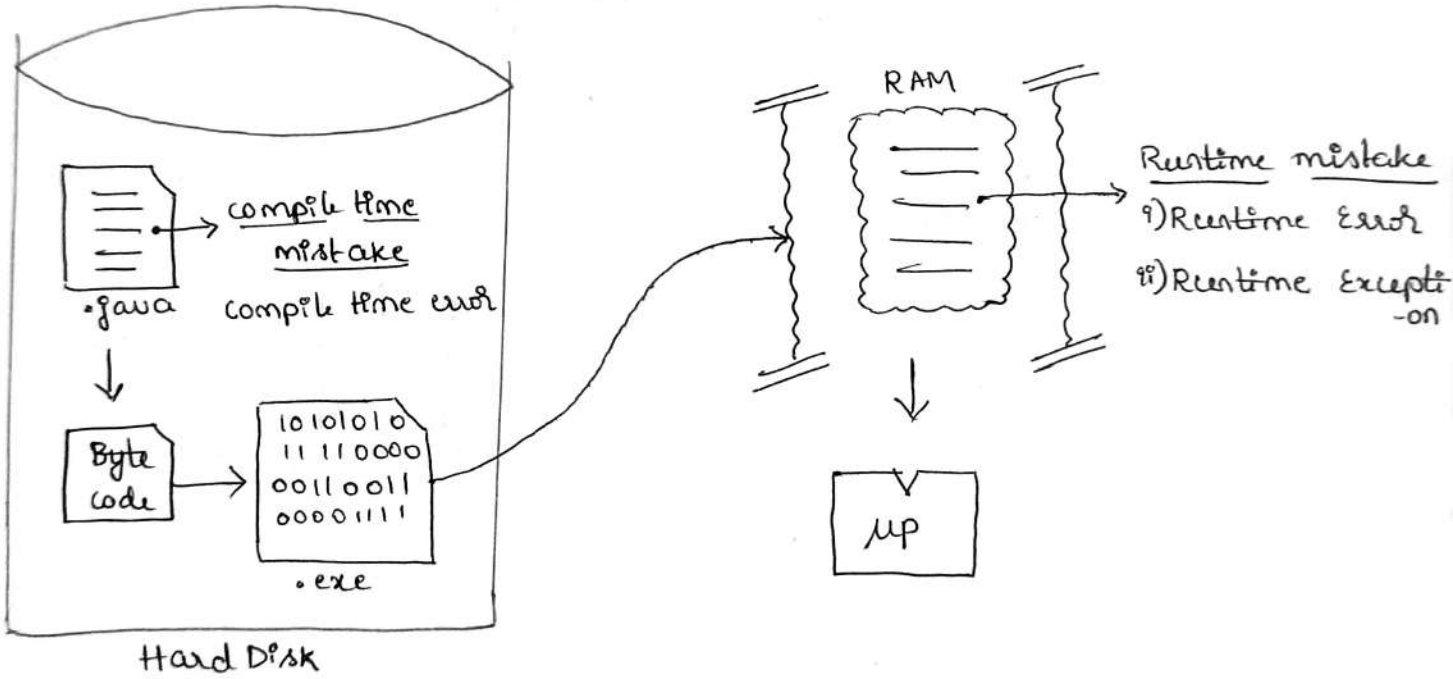
}

Java

1/9/23

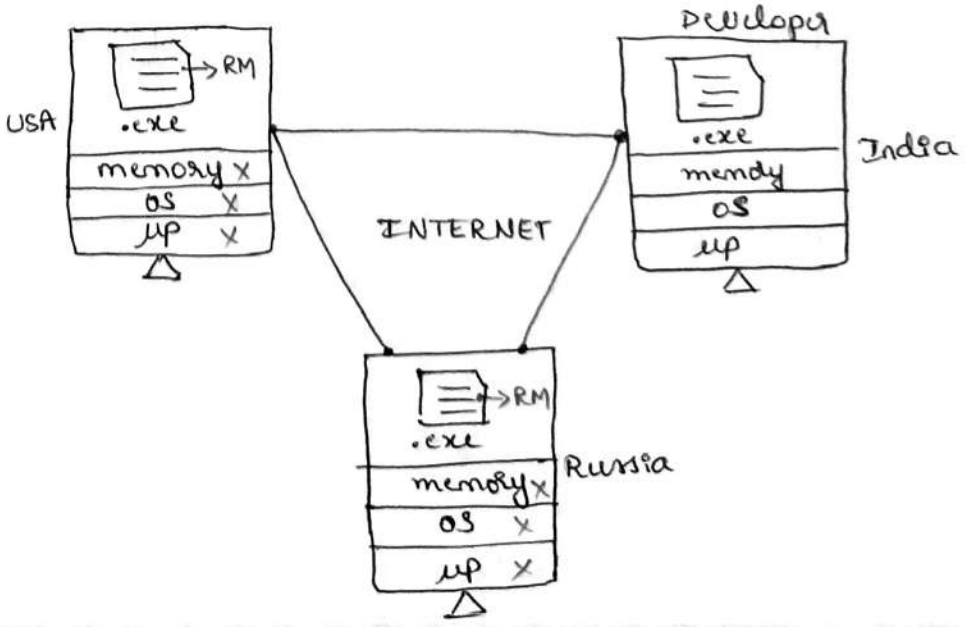
Exception Handling:

- Mistakes that occur during compilation are called compile time errors
- Mistakes that occur during execution (runtime) are classified into Runtime error & Runtime exception.
- If the Runtime mistakes are not handled, they can damage the memory, OS, (or) hardware of the computer.
- In today's web-based world if software products produce Runtime mistakes which are not handled within the software itself, it can damage the customers system & cause consequences for developing company.



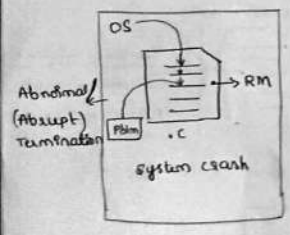
⇒ 1990's - 2000's

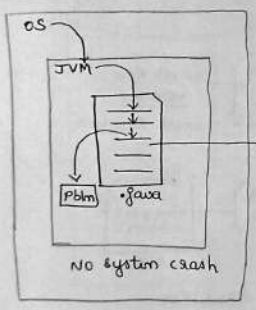
INTERNET
(web based appn)



Note:

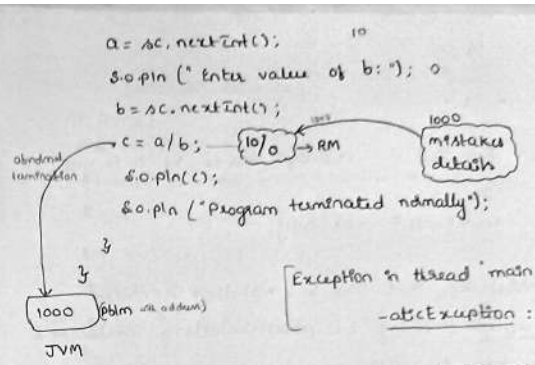
- Whenever there is a runtime mistake, the control immediately exits the pgm. This is called as abnormal termination.
- In C, the control along with the pblm will fall into the hands of OS which will not be able to handle it. This may result in a system crash.
- In Java, the control along with the pblm falls into the hands of JVM which is capable of handling Runtime mistakes in Java. Hence there is no system crash. This is why Java is called a Robust language.
- The only responsibility of the programmer now is to ensure pgm doesnot terminate abnormally.





Eg: 1 Unhandled RM (runtime mistake) leading to abnormal termination:

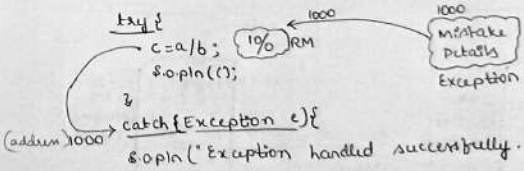
```
import java.util.Scanner;
class Demo {
    p s v m(){
        Scanner sc=new Scanner(System.in);
        int a, b, c;
        S.o.pln("Enter value of a:");
        a=sc.nextInt();
        S.o.pln("Enter value of b:");
        b=sc.nextInt();
        c=a/b;
        S.o.pln(c);
        S.o.pln("Program terminated normally");
    }
}
```



- Dividing by zero is not allowed in programming & leads to Runtime mistake.
- In the above eg, the runtime mistake is not handled hence pgm terminated abnormally.

Eg-2: Most Runtime mistakes can be handled in java using the try - catch block.

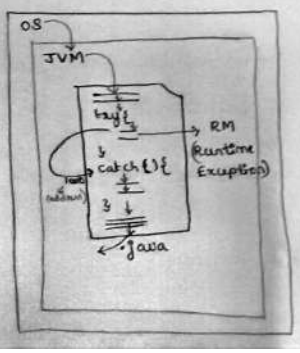
```
import java.util.Scanner;
class Demo1
    p s v m(){ int a, b, c;
        Scanner sc=new Scanner(System.in); S.o.pln("Enter value of a:");
        a=sc.nextInt();
        S.o.pln("Enter value of b:");
        b=sc.nextInt();
        try{
            c=a/b;
            S.o.pln(c);
        }
        catch(Exception e){
            S.o.pln("Exception handled successfully. Divide by 0 is
                not possible");
        }
        S.o.pln("Program terminated normally");
    }
}
```

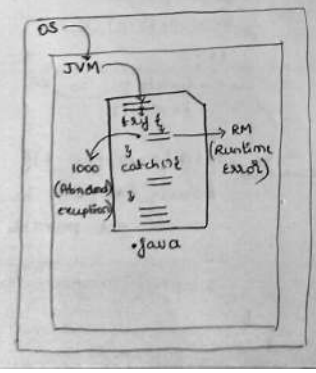


o/p: Enter value of a:
10
Enter value of b:
0
Exception handled successfully. Divide by 0 is not possible.
Program terminated normally.

- A Runtime mistake that can be handled is called Runtime Exception. (try & catch) Using Exception handling mechanisms we can ensure pgm terminates normally if we get a runtime exception.
- A Runtime mistakes which cannot be handled are called Runtime Errors. If a runtime error occurs, the pgm will terminate abnormally even if we use Exception handling mechanisms.

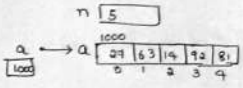
Exception Handling: It is a process to ensure the pgm dosnot abnormally terminate & follows the normal control flow in the occurance of runtime exception using Exception handling mechanisms like try-catch, finally blocks.

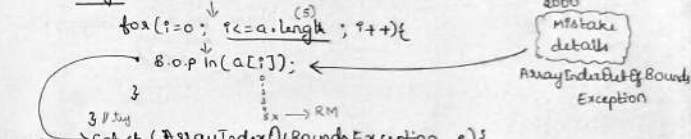




eg-3:

```
import java.util.Scanner;
class Demo{
  p s. vm(){
  Scanner sc = new Scanner(System.in);
  S.o.pln("Enter size of array: ");
  int n = sc.nextInt();
  int a[] = new int[n];
  S.o.pln("Enter elements of the array:");
  int i;
    for(i=0; i<a.length-1 ; i++){
    a[i]= sc.nextInt();
  }
  S.o.pln("Elements of the array are:");
  try{
      for(i=0; i<=a.length ; i++){
      S.o.pln(a[i]);
  }
  } // try
  catch(ArrayIndexOfBoundsException e){
      S.o.pln("Buffer overrun");
  }
      S.o.pln("program terminated normally");
  }
}
```





- The above pgm works even if the catch block is written as follows:

```
catch(Exception e){
    S.o.pln("Buffer Overrun");
}
```

- This means that e which is of type Exception is able to collect address of object of ArrayIndexOutOfBoundsException. This is possible only if AIOOBE is a child of Exception.

i.e class Exception{
}
class ArithmeticException extends Exception{
}
class ArrayIndexOutOfBoundsException extends Exception{
}

Note: Exception e = 2000(A100BE) ✓ (Parent type reference can hold the address of child class object)

→ class Student{
}
class Person{
}
Student s = new Person(); ✗ [Error]

→ class Plane{
}
class Cargo extends Plane{
}
Plane p = new Cargo(); ✓

(Plane) ← extends (Cargo)

Q. Write a note on Ducking. 4/9/23

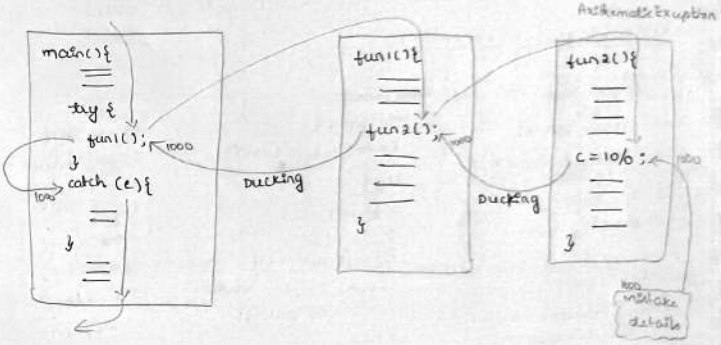
The process in which the called() method() doesnot handle the exception that occurs & passes it on to the caller is called Ducking.

Note: Whenever there is an exception within a called() method & it is not handled (No try-catch block) then the pgm doesnot terminate abruptly & after the control looks for try-catch block in the caller. The pgm will terminate abruptly only when there is no try-catch block in any of the caller.

eg:
```
class Demo{
  P.S.V.m(){
    Demo d1 = new Demo();
    try{
      d1.fun1();
    }
    catch(Exception e){
      S.o.pln("Exception handled in main()");
    }
    S.o.pln("Pgm terminated normally");
  }
}
```

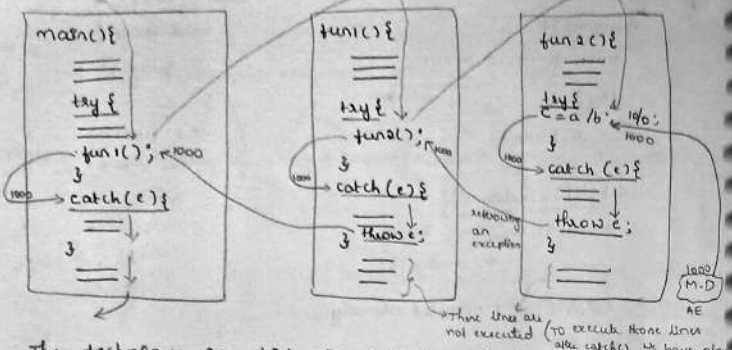
```
class Demo{
  void fun1(){ //called method
    fun2(); //caller
  }
  void fun2(){ //called method
    int a,b,c;
    a=10;
    b=0;
    c=a/b;
    S.o.pln(c);
  }
}
```

10/0 Mistake details ArithmeticException

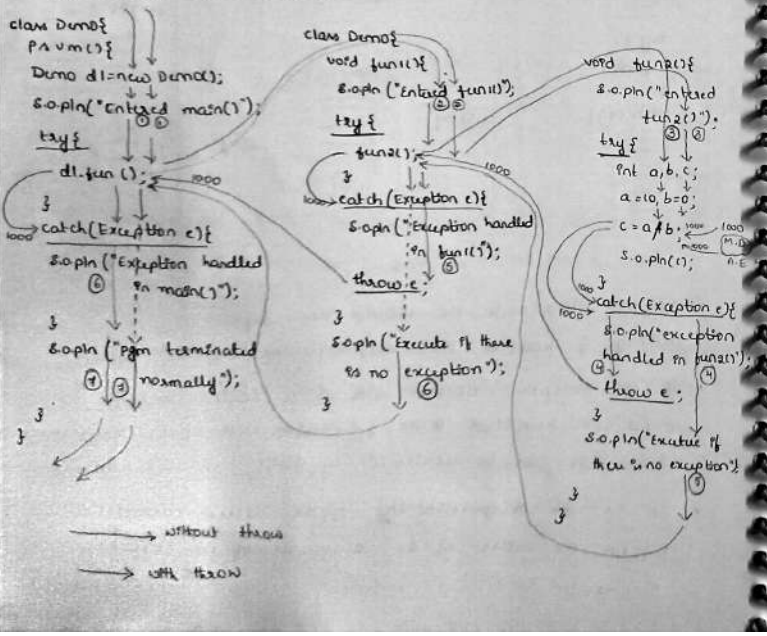


- As per standards we should not depend on Ducking & handle the exception wherever it occurs.
- If an exception occurs within a called method, it must be handled there & within each of the callers as well. This can be achieved by using throw keyword.
- It is the responsibility of the called method to inform the caller of the occurance of an exception.

throw → used to throw the address of exception object to the caller



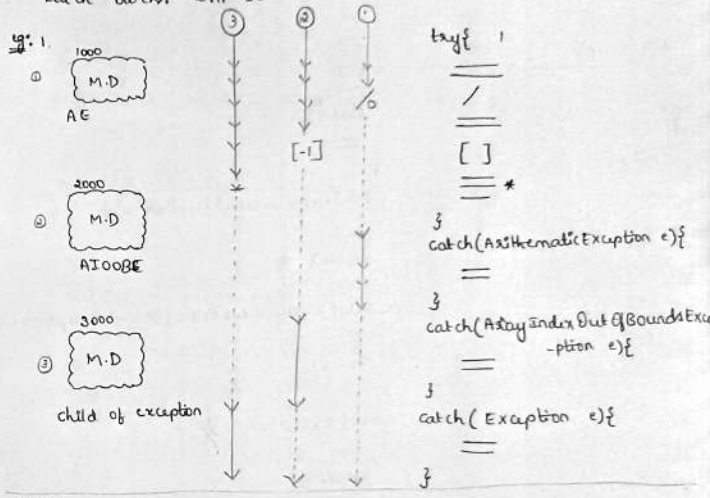
The technique in which the called method throws the exception to the caller after handling it is called Rethrowing an exception.

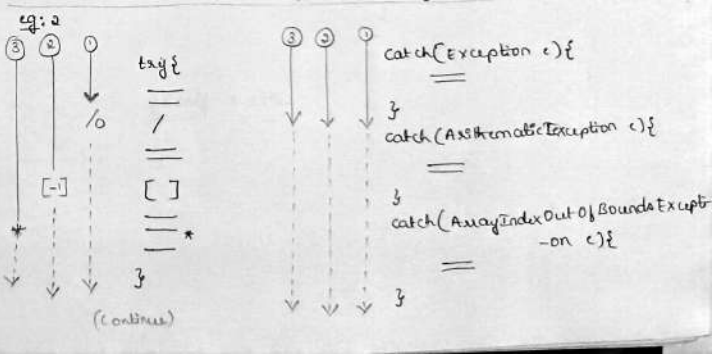


5/9/23

(Q) can one try block have multiple catch blocks?

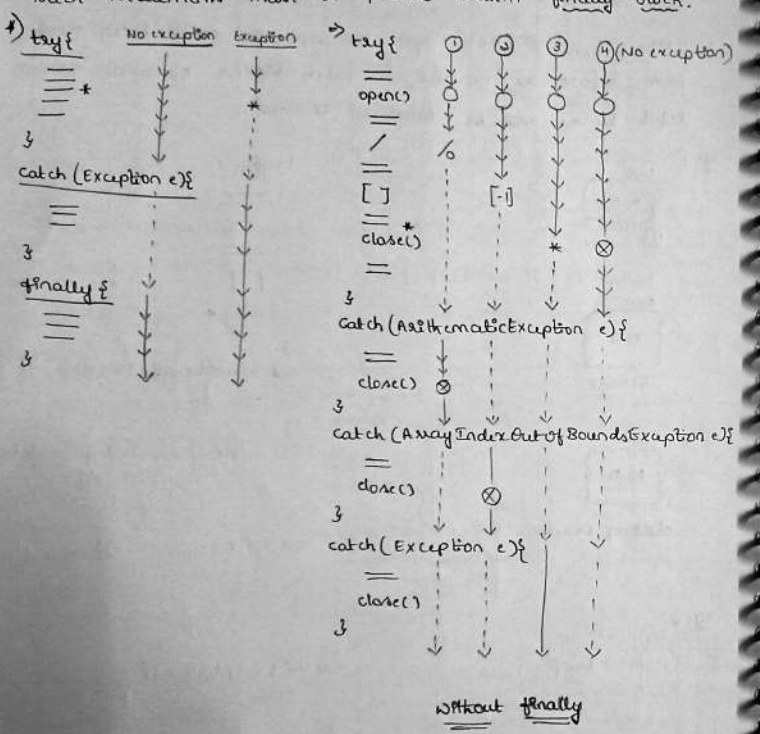
- One try block can have multiple catch blocks but only one catch block will be executed based on the exception that occurs.
- Here, we must note that the specific catch blocks must come before the generalized catch blocks otherwise specific catch blocks will be rendered useless.

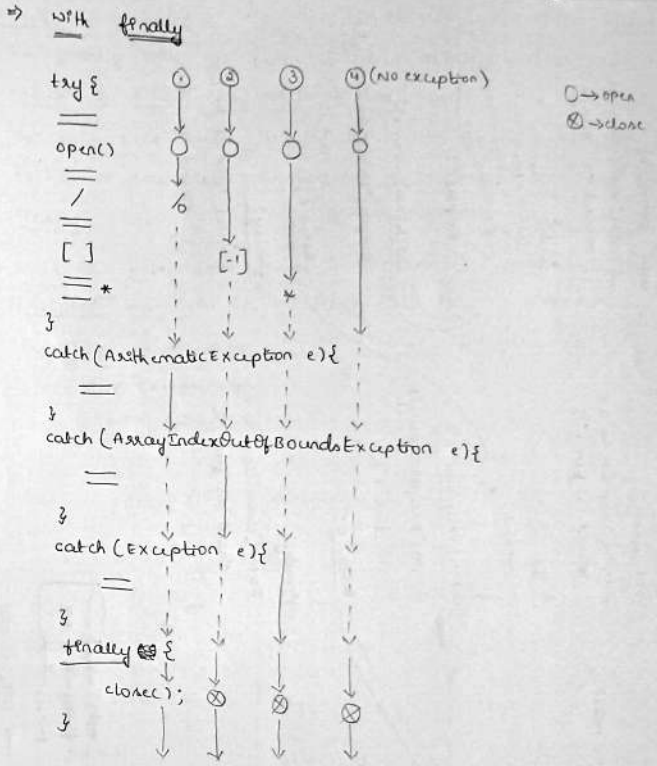




Finally block:

If there are a set of statements that must be executed irrespective of whether an exception occurs (or) not. then, such statements must be placed within finally block.





Note:

6/9/23

- The finally block will get executed even if the control encounters return & throw keywords before reaching the finally block.
- The only case where finally block will not be executed is if it encounters System.exit(0); before reaching the finally block.
  - System.exit(0) → normal termination
  - System.exit(-1) → abnormal
- When the exception object is created, the stack trace & exception msg will be bundled into it. We can retrieve them by using printStackTrace() & getMessage()

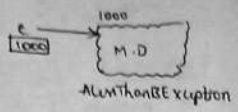
```
=> class Exception{
       void printStackTrace(){
           ==
       }
       Public String getMessage(){
           ==
       }
   }
```

```
=>
main(){
  ==
  try{
    ==
    fun1();
    ==
  }
  catch (Exception e){
    ==
    e.printStackTrace();
    S.o.pln(e.getMessage());
  }
  ==
}
```

```
fun1(){
  ==
  try{
    ==
    fun2();
    ==
  }
  catch (Exception e){
    ==
    throw e;
  }
  ==
}
```

```
fun2(){
  ==
  try{
    ==
    /0/[-1]/*
    ==
  }
  catch (Exception e){
    ==
    throw e;
  }
  ==
}
```

```
try{
  if(a>b){
    c=a-b;
    s.o.pln(c);
  }
  else{
    AlesThanBException e = new AlesThanBException();
      (Manually created exception object)
    throw e; (Manually throwing exception)
  }
}
catch(AlesThanBException e){
  s.o.pln("Exception handled successfully");
  s.o.pln(e.getMessage());
}
s.o.pln("Exiting main");
}
}
```



throw new AlesThanBException();

Note: ⑧ can object of any class be thrown using throw keyword ?

↳ No. only objects of classes that are directly / indirectly children of throwable class can be thrown using throw keyword.

Throwable
↑ extends
Exception

7/9/23

⇒ Write a pgm that throws NegativeBalanceException when the amount entered is negative & minimum balance exception when amount entered is greater than available balance. (eg of custom exception)

```
-> import java.util.Scanner;
class NegativeBalanceException extends Exception{
  public String getMessage(){
    return "You have entered a negative amount, withdrawal
            not possible";
  }
}
class MinimumBalanceException extends Exception{
  public String getMessage(){
    return "Amount entered is greater than available balance
                                                  -e";
  }
}

class Bank{
  int bal, amt;
  Bank(int b){ (parameterized constructor)
    bal = b;
  }
  void input(){
    Scanner sc=new Scanner(System.in);
    s.o.pln("Enter amount to be withdrawn : ");
    amt = sc.nextInt();
  }
  void withdraw(){
    try{
      if(amt<0){
        throw new NegativeBalanceException();
      }
```

```
else if (amt > bal){
    throw new MinimumBalanceException();
}
else {
    S.o.pln ("Withdrawal successful");
    bal = bal - amt ;
    S.o.pln ("Available balance =" + bal);
}
}
catch (NegativeBalanceException e) {
    S.o.pln (e.getMessage());
}
catch (MinimumBalanceException e){
    S.o.pln (e.getMessage());
}
} // withdraw()
} // Bank

Class BankDemo{
    P s v m(){
        Bank b1 = new Bank (1000);   // parameterized constructor
        b1.input();
        b1. withdraw();
    }
}
```

throws keyword :

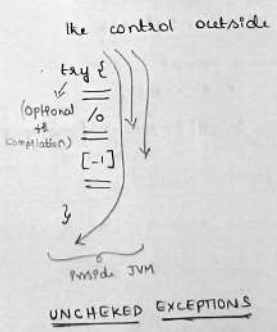
- The throws keyword is used to declare the exceptions that can be thrown by a method().
- It is not a compulsion (compulsory) to declare exception in the signature of a method using throws but we must do it as per standards.

eg:

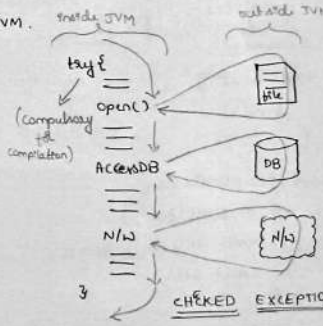
```
class Demo{
    void fun1() throws ArithmaticException, ArrayIndexOutOfB
                                            -oundsException
    {
        try{
            ==
            /0
            ==
            [-1]
            ==
        }
        catch(ArithmaticException e){
            ==
            throw e;
        }
        catch(ArrayIndexOutOfBoundsException e){
            ==
            throw e;
        }
    }
```

Hierarchy of Exception classes :

- There are some statements that takes the control outside the purview / supervision of the JVM. Such statements must be placed within try block. otherwise compilation will fail.
- eg: File operations, DB operations, Network operations take the control outside JVM.

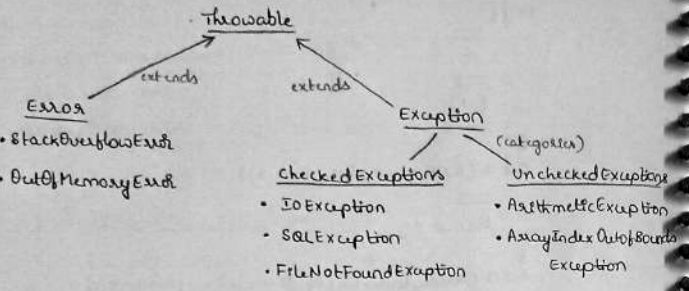


UNCHECKED EXCEPTIONS



CHECKED EXCEPTIONS

- The Exceptions that occur due to control being taken outside the pgm are called checked Exceptions.



Note: UNCHECKED EXCEPTIONS :

If we don't place the exceptions that occurs within try block then the pgm will be terminated abnormally & falls in the hands of JVM along with Exception.

Here pgm will start & also possible for compilation but we cannot execute as it is Runtime Exception.

CHECKED EXCEPTIONS :

If the Exception occurs outside the purview of JVM (i.e file/DB/n/w) & if we don't place them inside try block then pgm will not start & compilation will not take place. (so it is compulsory to use try block)

StackOverflowError eg:

```
class Demo{
  void fun(){
    fun();
  }
}
class ExceptionDemo{
    P S V m(){
      Demo d1 = new Demo();
       d1.fun();
  }
}
```

OutOfMemoryError

```
class Demo{
    P S V m(){
  int a[] = new int[999999999];
    }
}
```

8/9/23

Note :

- On the occurance of the Runtime exception the pgm will terminate, even if the statements are placed within try block But in the presence of finally block it will get executed before termination of the pgm.

eg:
```
class Demo{
    P S V m(){
      S.o.pln("Entering main()");
      try{
        S.o.pln("Entered try block");
        int a[]=new int[999999999];
      }
      catch(Exception e){
        S.o.pln("Executed catch block");
      }
      finally{
        S.o.pln("Finally block executed");
      }
      S.o.pln("Pgm terminated normally");
    }
}
```

o/p:
```
Entering main()
Entered try block
Finally block executed
  → Exception in thread "main" java.lang.OutOfMemoryError : Java heap space
```

Overriding & Exception Handling :

LISKOV substitution principles :

1) If the parent method throws an exception, the child method need not throw an exception.

eg: class Parent{
    void fun1() throws IOException{
    ==
    }
}
class child extends Parent{
    void fun1(){
    ==
    }
}

ii) The child method may throw the same exception as the parent method.

eg: class Parent{
    void fun1() throws IOException{
    ==
    }
}
class child extends Parent{
    void fun1() throws IOException{
    ==
    }
}

iii) If the parent method throws a checked exception then a child method can throw different exception only if it is related & narrower than the parent exception.

eg: 1 class parent{
    void fun1() throw IOException{
    ==
    }
}
class child extends Parent{
    void fun1() throws FileNotFoundException
    {
    ==
    }
}

(Parent) IOException [Broader]
↑
FileNotFoundException (child) [Narrower]

eg: 2 class parent{
    void fun1() throws FileNotFoundException{
    ==
    }
}
class child extends Parent{
    void fun1() throws IOException{
    ==
    }
}

iv) If parent & child methods must throw unrelated exceptions if is possible if exceptions of unchecked.

eg: class Parent{
    void fun1() throws ArithmeticException{
    ==
    }
}
class child extends Parent{
    void fun1() throws ArrayIndexOutOfBoundsException{
    ==
    }
}

(Separate type of method)

=> Write a pgm to override toString() & equals() of the object class.

- By default toString() method returns address of the object in hexadecimal format.
- By default equals() method compares address of two objects.
- In string class equals() method has been overridden to compare content of the strings.
- In StringBuffer & StringBuilder classes equals() method is not overridden. This implies address of a objects will get compared when equals() is called.

eg: class Parent{
    void fun1() throws IOException{
        ==
    }
}
class child extends Parent{
    void fun1(){
        ==
    }
}

ii) The child method may throw the same exception as the parent method.

eg: class Parent{
    void fun1() throws IOException{
        ==
    }
}
class child extends Parent{
    void fun1() throws IOException{
        ==
    }
}

iii) If the parent method throws a checked exception then a child method can throw different exception only if it is related & narrower than the parent exception.

eg: 1 class parent{
    void fun1() throws IOException{
        ==
    }
}
class child extends Parent{
    void fun1() throws FileNotFoundException
    {
        ==
    }
}

(Parent) IOException [Broader]
↑
FileNotFoundException (child) [Narrower]

eg: 2 class parent{
    void fun1() throws FileNotFoundException{
        ==
    }
}
class child extends Parent{
    void fun1() throws IOException{
        ==
    }
}

iv) If parent & child methods must throw unrelated exceptions if is possible if exceptions are unchecked.

eg: class Parent{
    void fun1() throws ArithmeticException{
        ==
    }
}
class child extends Parent{
    void fun1() throws ArrayIndexOutOfBoundsException{
        ==
    }
}

(Separate type) method

=> Write a pgm to override toString() & equals() of the object class.

- By default toString() method returns address of the object in hexadecimal format.
- By default equals() method compares addrss of two objects.
- In string class equals() method has been overridden to compare content of the strings.
- In StringBuffer & StringBuilder classes equals() method is not overridden. This implies address of 2 objects will get compared when equals() is called.

*) (syntax type) class Object { → Root node / Master node

:

Public String toString {

return "Address of object" ;

}

Public boolean equals (Object obj){

==

compare address of two objects

==

return true/false ;

}

:

}

object
↑
Parent
↑
child

*) class Person{

int age ;

Person (int age){

this.age =age ;

}

}

class PersonDemo{

P S V m(){

Person p1 = new Person (10);

Person p2 = new Person (10);

S.o.pln (p1); → Address of 1st object (1000) — (p1.toString)

S.o.pln (p2); → Address of 2nd object (2003) — (p2.toString)

S.o.pln (p1.equals(p2)); → False

}

}

p1,p2 by default

↓

holds the address of current object.

so it is retrived by using this keyword

Soln : Overriding

*) class Person{

int age;

Person (int age){

this.age =age;

}

public String toString(){ → ValueOf → static bcz of class name

return String.valueOf(age); String (method declared by using class name → static)

}

public boolean equals(Object obj){ p2

Person tp =(Person)obj; → downcasting

if(tp.age == this.age){

p2 p1

return true;

}

else{

return false;

}

}

(parent) object ↑ Person (child) — downcasting

class PersonDemo{

P S V m(){

Person p1 = new Person(10);

Person p2 = new Person(10);

S.o.pln(p2);

S.o.pln(p2);

S.o.pln(p1.equals(p2));

}

}

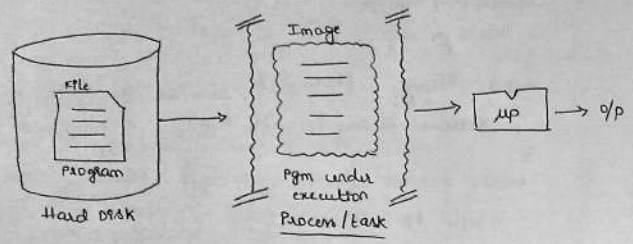
o/p: 10 — (p1)

10 — (p2)

true. — (content of p1 & p2 compared)

Multithreading in Java:

- A pgm under execution is called a process / task.



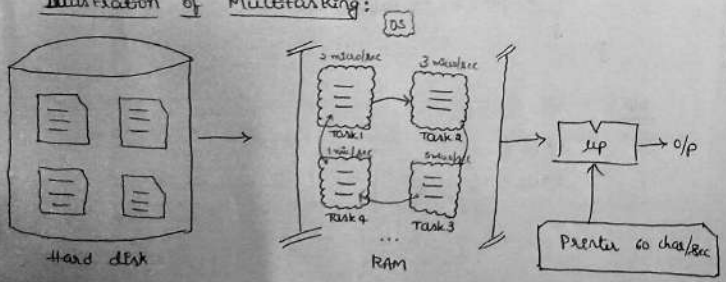
Multitasking:

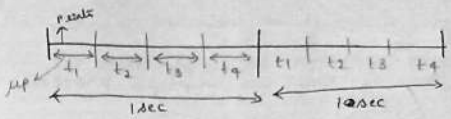
- Executing multiple tasks simultaneously is called multitasking.
- It is performed to improve the efficiency of microprocessor.
- In reality µp can execute only one instruction at a time. Hence simultaneous execution is only an illusion created by extremely fast context switching.

Context Switching:

Transferring the control from one task to another is called Context Switching.

Illustration of Multitasking:



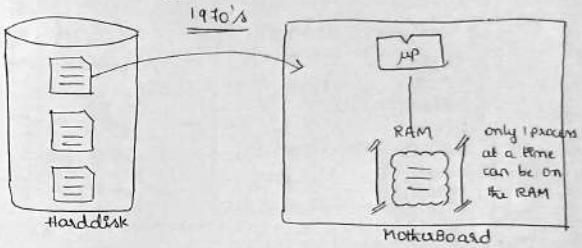


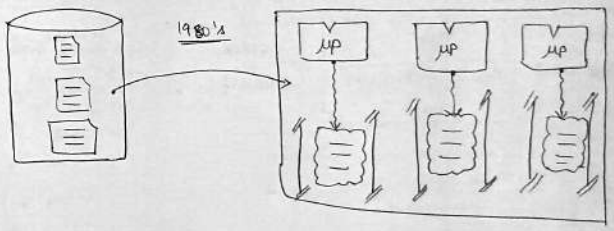
12/9/23

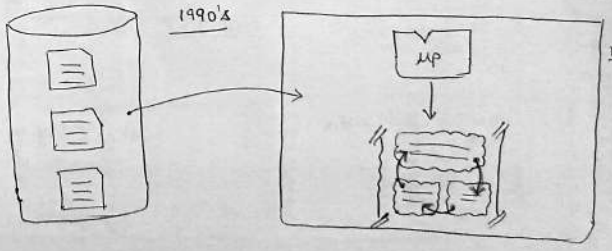
Note:

- How much time must be allocated for which process is decided by the OS.
- Context switching is performed by the OS.

Structural changes in computers:

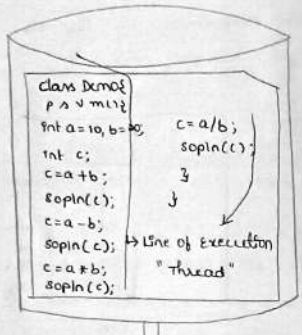


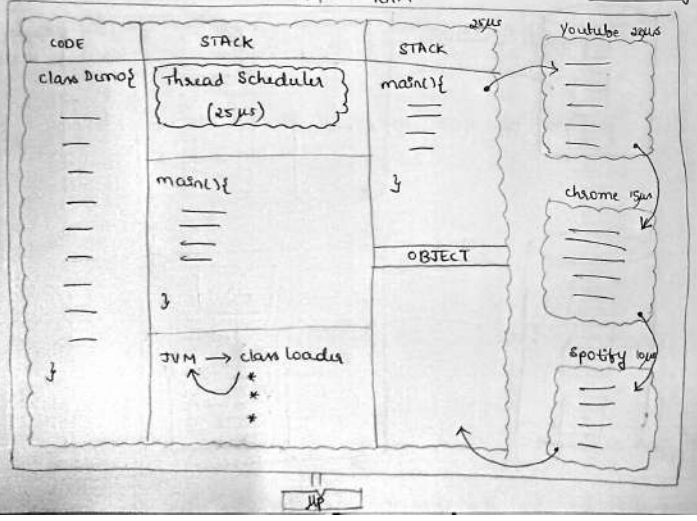




Thread Scheduler :

- Once the OS allocates the time for multiple processes it is collected by the thread scheduler.
- This means that the time management of a Java pgm is controlled & co-ordinated by the thread scheduler.





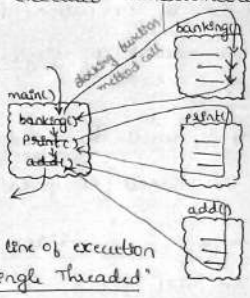
- The line of execution is technically called as a thread.
- In the above eg, the pgm has a single line of execution hence it is single-threaded pgm.

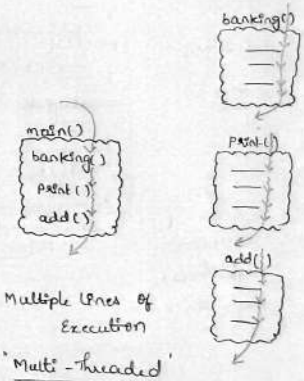
Introduction to Multi-Threading :

- The technique of writing a pgm with multiple lines of execution is called as multi-threading.
- While designing applications if there are independent sub-tasks, then they must be executed simultaneously.

eg-1 :

```
class Task{
  void banking(){
    Scanner sc=new Scanner(System.in);
    S.o.pln("Enter username : ");
    String name = sc.next();
    S.o.pln("Enter the PIN:");
    int pin = sc.nextInt();
    S.o.pln("Login successful");
  }
  void print(){
    int i;
    for(i=1; i<=5; i++){
      sopln(i);
      Thread.sleep(5000);
    }
    catch (Exception e){
      sopln(e.getMessage());
    }
  }
  void add(){
    int a=1000, b=2000;
    int c=a+b;
    sopln(c);
  }
}
class Demo{
  psvm(){
    Task t=new Task();
    t.banking();
    t.print();
    t.add();
  }
}
```

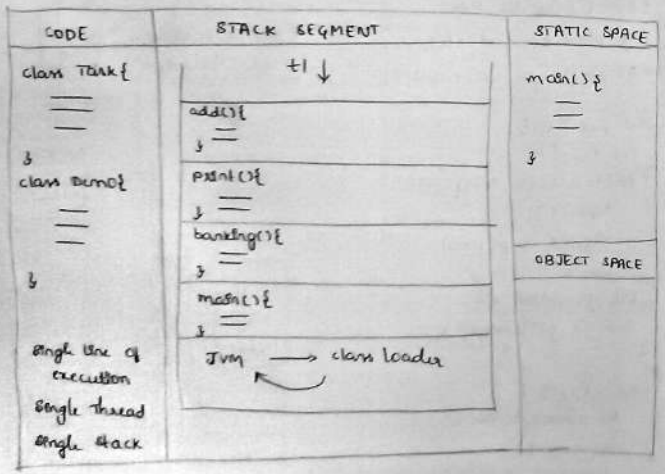




- In the above pgm, banking(), print() & add() are independent sub-tasks hence must be executed simultaneously. But as there is only a single line of execution, we are making one task to wait for another task to finish its execution. Even though o/p of one task is not required for another task.

Note:

- 1 line of execution (thread) is allocated for 1 stack. This means a single-threaded pgm will have only one stack.
- In a multi-threaded pgm, multiple stacks are created & a thread is given for each stack. (eg. 2)



13/9/23

Steps involved in creating Multi-threaded pgm:

Step 1: Logic of the independent task must be placed within run() method.

Step 2: The run() method must be placed within a class that extends Thread class.

Step 3: Within the constructor of a class make a call to the start() method.

Note:

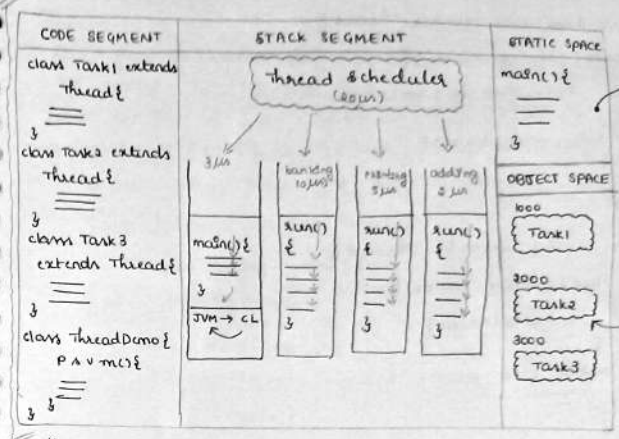
The two jobs of start() method are:

- creates a new stack whenever it is called.
- Internally calls run() method & place its activation record on the newly created stack.

eg-1:

```
class Task1 extends Thread{
    Task1(){                    // constructor
      start();
    }
    public void run(){
      Scanner sc = new Scanner(System.in);
      S.o.pln("Enter Username:");
      String name = sc.next();
      SOpln("Enter PIN:");
      int pin = new sc.nextInt();
      S.o.pln("Login Successful");
    }
}
class Task2 extends Thread{
    Task2(){
      start();
    }
    public void run(){
```

```
        int i;
        try{
            for(i=1; i<=5; i++){
                sopln(i);
                Thread.sleep(1000);
            }
        }
        catch(Exception e){
            sopln(e.getMessage());
        }
    }
}
class Task3 extends Thread{
    Task3(){
        start();
    }
    public void run(){
        int a=1000;
        int b=2000;
        int c=a+b;
        sopln("Addition Result="+c);
    }
}
class ThreadDemo{
    P.S.V.m(){
        Task1 t1=new Task1();
        Task2 t2=new Task2();
        Task3 t3=new Task3();
    }
}
```



- run() method can be called manually. It will not result in compile-time error. But the pgm will remain a single-threaded pgm.
- There is no compulsion that start() must be placed within constructor. It can be called wherever we want. But placing it in the constructor will create a new thread as soon as object is created.
- The thread is not active after creating the object. It becomes active only when the start() method is called.

Naming of Threads:

- In order to give the name for a thread, the name must be passed to the constructor of thread-class (or) we must use the setName() method.
- In order to read the name of a thread, we can use getName() method.

eg:

(example-1)

```
class Task1 extends Thread{
    Task1(String name){   // name is provided for Task1 which is child class of Thread.
        super(name);      // → using super, we pass (provided) that name to (parent class) Thread.
    }
    public void run(){
        ==
        ==
    }
}
class Task2 extends Thread{
    Task2(String name){
        super(name);
    }
    public void run(){
        ==
        ==
    }
}
class Task3 extends Thread{
    Task3(){
        start();
    }
    public void run(){
        ==
        ==
    }
}
class ThreadDemo{
    p s v m(){
        Task1 t1 = new Task1("BankingThread");   → Naming of thread by passing parameter to constructor. Here, the thread name is given to Task1
        Task2 t2 = new Task2("PrintingThread");
        Task3 t3 = new Task3(" );
        t1.start();
        t2.start();
        t3.setName("Adding Thread");   → naming of thread by using setName()
        So.pln(t1.getName());   // BankingThread
        So.pln(t2.getName());   // PrintingThread
        So.pln(t3.getName());   // AddingThread
    }
}
```

Note:

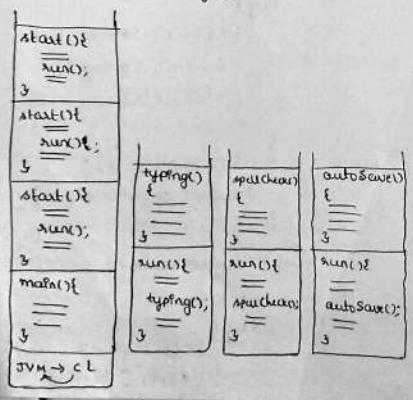
- Without using address variable we can get thread name by using `Thread.currentThread.getName();`

⇒ Write a pgm to achieve Multi-Threading of independent tasks using only one run() method.

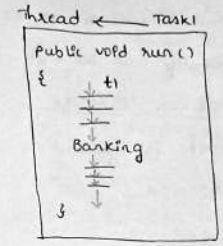
(example-2)

```
class MSWord extends Thread{
    MSWord(String name){
        super(name);
    }
    public void run(){
        if(getName().equals("Typing")){
            typing();
        }
        else if(getName.equals("Checking")){
            spellCheck();
        }
        else{
            autoSave();
        }
    }
    void typing(){
        Int i;
        try{
            for(i=1; i<=10; i++){
                So.pln("Typing");
                Thread.sleep(3000);
            }
        }
        catch(Exception e){
            So.pln(e.getMessage());
        }
    }
    void spellCheck(){
        Int i;
        try{
            for(i=1; i<=5; i++){
                So.pln("Checking spelling mistakes");
```

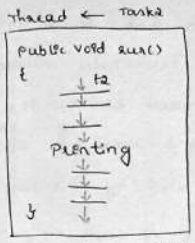
```
        Thread.sleep(5000);
    }
}
catch(Exception e){
    S.o.pln(e.getMessage());
}
}
void autoSave(){
    int i;
    try{
        for(i=1; i<=5; i++){
            S.o.pln("Saving ....");
            Thread.sleep(6000);
        }
    }
    catch(Exception e){
        S.o.pln(e.getMessage());
    }
}
class ThreadDemo{
    P S V m(){
        MSWord m1 = new MSWord("Typing");
        MSWord m2 = new MSWord("checking");
        MSWord m3 = new MSWord("Saving");
        m1.start();
        m2.start();
        m3.start();
    }
}
```



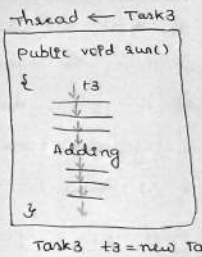
Example-1 : 14/9/23



Task1 t1 = new Task1();
t1.start();



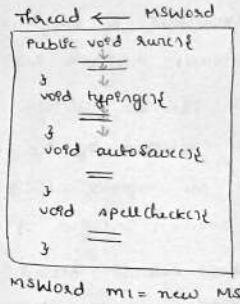
Task2 t2 = new Task2();
t2.start();



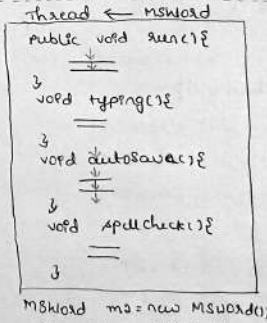
Task3 t3 = new Task3();
t3.start();

Example-2:

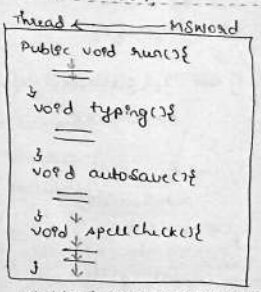
```
class MSWord extends Thread
{
  public void run(){
    ==
  }
  void typing(){
    ==
  }
  void autoSave(){
    ==
  }
  void spellCheck(){
    ==
  }
}
```



MSWord m1 = new MSWord();
m1.setName("Typing");
m1.start();



MSWord m2 = new MSWord();
m2.setName("Saving");
m2.start();



MSWord m3 = new MSWord();
m3.setName("checking");
m3.start();

Daemon threads:

If there are threads that must begin execution at the start of the pgm & terminate automatically at the end of the pgm, they must be made Daemon threads. Background tasks that serve & support foreground tasks must be executed by Daemon threads.

There are 2 types of threads in Java:

i) User thread ii) Daemon thread.

The pgm terminates when all user threads have finished its execution & only Daemon threads are left. The Daemon threads are killed by the JVM.

How to create Daemon Threads?

- Logic executed by the Daemon thread must be placed within an infinite loop.
- The Daemon status of the thread must be true.

Eg:
```
class MSWord extends Thread {
    public void run(){
        if(getName().equals("Typing")){
            typing();
        }
        else if(getName().equals("checking")){
            spellcheck();
        }
        else{
            autoSave();
        }
    }
    void typing(){
        int i;
        try{
            for(i=1; i<=20; i++){
                S.o.pln("Typing");
                Thread.sleep(1000);
            }
        }
        catch(Exception e){
            S.o.pln(e.getMessage());
        }
    }
    void autoSave(){
        try{
            for(;;){ // infinite loop
                S.o.pln("saving...");
                Thread.sleep(2000);
            }
        }
        catch(Exception e){
            S.o.pln(e.getMessage());
        }
    }
    void spellCheck(){
        try{
            for(;;){ // infinite loop
                S.o.pln("checking spelling mistakes");
                Thread.sleep(2000);
            }
        }
        catch(Exception e){
            S.o.pln(e.getMessage());
        }
    }
}
class ThreadDemo{
    p s v m(){
        MSWord m1 = new MSWord();
        MSWord m2 = new MSWord();
        MSWord m3 = new MSWord();
        m1.setName("Typing");
        m2.setName("saving");
        m2.setDaemon(true);
        m3.setName("checking");
        m3.setDaemon(true);
        m1.start(); m2.start();
        m3.start();
    }
}
```

saving, checking → Background tasks (Daemon thread)

Typing → foreground task (User thread)

Ways of creating threads in Java:

i) By extending thread class.

ii) By implementing Runnable interface.

eg: for creating threads by implementing threads Runnable interface:

①

```
import java.util.Scanner;
class Task1 implements Runnable{
    Public void run(){
        ==
    }
}
class Task2 implements Runnable{
    Public void run(){
        ==
    }
}
class Task3 implements Runnable{
    Public void run(){
        ==
    }
}
class ThreadDemo{
  P s v m(){
    Task1 t1 = new Task1();   → Thread th1 = new Thread(t1);
    Task2 t2 = new Task2();   → Thread th2 = new Thread(t2);
    Task3 t3 = new Task3();
    Thread th3 = new Thread(t3);
     th1.start();
     th2.start();
     th3.start();
  }
}
```

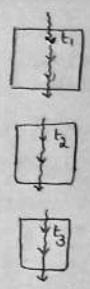
eg: ②

Implementing Runnable interface using constructor

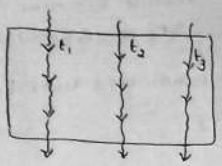
```
import java.util.Scanner;
class Task1 implements Runnable{
    Task1(){
      Thread th1 = new Thread(this);
       th1.start();
    }
    Public void run(){
        ==
    }
}
class Task2 implements Runnable{
    Task2(){
      Thread th2 = new Thread(this);
       th2.start();
    }
    public void run(){
        ==
    }
}
class Task3 implements Runnable{
    Task3(){
      Thread th3 = new Thread(this);
       th3.start();
    }
    Public void run(){
        ==
    }
}
class ThreadDemo{
  P s v m(){
    Task1 t1 = new Task1();
    Task2 t2 = new Task2();
    Task3 t3 = new Task3();
  }
}
```

21/9/23

Multithreading case 1:



Multithreading case 2:



Case-2 eg:

```
class ATM implements Runnable{
    ATM(){
        Thread t1 = new Thread(this);
        Thread t2 = new Thread(this);
        Thread t3 = new Thread(this);
        t1.setName("Person 1");
        t2.setName("Person 2");
        t3.setName("Person 3");
        t1.start();
        t2.start();
        t3.start();
    }
    Public void run(){
        S.o.pln(Thread.currentThread().getName() + "has entered ATM");
        S.o.pln(Thread.currentThread().getName() + "has inserted ATM card");
        S.o.pln(Thread.currentThread().getName() + "has inserted Pin & withdrawing money");
        S.o.pln(Thread.currentThread().getName() + "has exited the atm");
    }
}
```

```
class Demo{
    P.S.V.m(){
        ATM a = new ATM();
    }
}
```

O/P: Person 1 has entered ATM
P2 has entered ATM
P2 has inserted atm card
P3 has entered atm
P3 has inserted atm card
P3 has entered pin & withdrawing money
P1 has inserted atm card
P1 has entered pin & withdrawing money
P1 has exited atm
P2 has entered pin & withdrawing money
P3 has exited atm.
P2 has exited atm.

jumbled o/p

Synchronization:

when multiple threads want to access the same resource & we want to ensure that only one thread can access the resource at a time then the resource must be synchronized.

Synchronization can be achieved in two ways:

i) By using synchronized keyword.
ii) By using join() method.

⇒ i) In the above pgm, if run() method is synchronized then o/p is as follows:

```
synchronized public void run(){
    ==
    ==
}
```

o/p: P1 has entered atm
P1 has inserted atm card
P1 has entered pin & withdrawing money
P1 has exited atm.
P3 has entered atm
P3 has inserted atm card
P3 has entered pin & withdrawing money
P3 has exited atm.
P2 has entered atm
P2 has inserted atm card
P2 has entered pin & withdrawing money
P2 has exited atm.

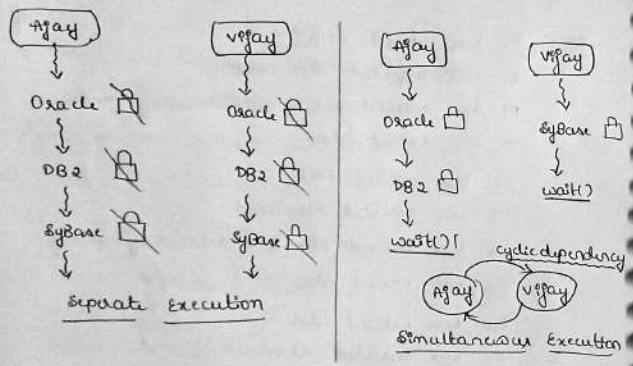
Using synchronized keyword the order of execution of threads cannot be predicted. This can be achieved by using join() method.

⇒ ii)
```
class ATM implements Runnable {
ATM() {
  Thread t1=new Thread(this);
  Thread t2=new Thread(this);
  Thread t3=new Thread(this);
  t1.setName("P1");
  t2.setName("P2");
  t3.setName("P3");
  try {
      t1.start();
      t1.join();
      t2.start();
      t2.join();
      t3.start();
}
catch(Exception e){
    S.o.println(e.getMessage());
  }
}
Public void run(){
    ==========
  }
}
class Demo{
   P s v m(){
     ATM a=new ATM();
   }
}
```

o/p: P1 has entered atm
P1 has inserted atm card
P1 has entered pin & withdrawing money
P1 has exited atm.
P2 has entered atm
P2 has inserted atm card
P2 has entered pin & withdrawing money.
P2 has exited atm
P3 has entered atm
P3 has inserted atm
P3 has entered pin & withdrawing money
P3 has exited atm.

Deadlocks in Java:

- An infinite wait condition is called a deadlock.
- Whenever a deadlock appears, termination of the pgm is the only solution.
- While designing applications, we must ensure that there is no cyclic dependency b/w 2 threads which is the primary reason for deadlocks.



Eg:

```
class Deadlock implements Runnable{
    String r1 = "Oracle";
    String r2 = "DB2";
    String r3 = "SyBase";
    Deadlock(){
        Thread t1 = new Thread(this);
        Thread t2 = new Thread(this);
        t1.setName("Afay");
        t2.setName("Vfay");
    }   t1.start();   t2.start();
```

```
Public void run(){
  if(Thread.currentThread.getName()).equals("Afay"){
      acquireAfayResources();
  }
  else{
      acquireVfayResources();
  }
}
void acquireAfayResources(){
    synchronized(r1){
      S.o.pln("Oracle acquired by Afay");
      synchronized(r2){
        S.o.pln("DB2 acquired by Afay");
        synchronized(r3){
          S.o.pln("SyBase acquired by Afay");
        }
      }
    }
}
void acquireVfayResources(){
    synchronized(r3){
      S.o.pln("SyBase acquired by Vfay");
      synchronized(r2){
        S.o.pln("DB2 acquired by Vfay");
        synchronized(r1){
          S.o.pln("Oracle acquired by Vfay");
        }
      }
    }
}
```

```
class Demo{
    P s v m(){
        Deadlock d = new Deadlock();
    }
}
```

22/9/23

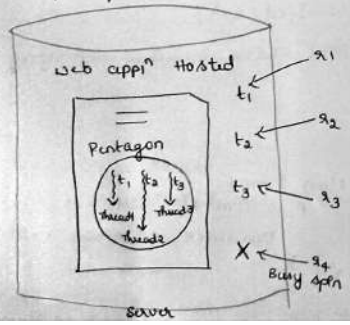
o/p-1: oracle acquired by Ajay
DB2 acquired by Ajay
SyBase acquired by vijay.

o/p-2: SyBase acquired by vijay
DB2 acquired by vijay
oracle acquired by Ajay.

(Q) write a note on Thread group in java.

In client-server architecture when a appln is hosted on the server it must be capable of processing multiple requests. The optimal approach would be to assign a thread for each request & process them simultaneously but creation of threads is a time consuming process. Hence if we create a thread upon arrival of request time taken to process it will increase effectively decreasing the efficiency of the system.

To address this from Java 1.4 the concept of thread-group was introduced. Here, we can create a bunch of threads before-hand & just assign the thread upon arrival of request to service it.




```
class Branch extends Thread {
    Branch (ThreadGroup tg, String name){
        super(name);
    }
    public void run(){
        ==
    }
}
class Demo{
    p s v m(){
        ThreadGroup tg = new ThreadGroup("Pentagon");
        Branch t1 = new Branch( tg, "Thread1");
        Branch t2 = new Branch (tg, "Thread2");
        Branch t3 = new Branch (tg, "Thread3");
        ==
        t1.start();
        t2.start();
        t3.start();
        ==
    }
}
```

When all the threads in the thread group have been utilized, further requests cannot be processed. This situation is called as Busy spin. A ExecutionRejectedException is generated in such case.

(Q) comment on the relationship b/w Thread class & Runnable interface.

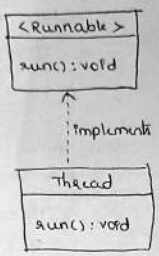
→ The Thread class internally implements Runnable interface
. The Runnable interface contains only run() method.

```
interface Runnable{
    void run();
}
```
Runnable interface

By default, public abstract

- Since Thread class is a child of Runnable interface, it must provide implementation for run() method. But it is an empty implementation as shown below.

```
class Thread implements Runnable{
    ==========
    public void run(){
    }
    ==========
}
```



- The reason for empty implementation is

i) Every class that extends Thread class will override run() method.

ii) To avoid Thread class from becoming abstract.

**** ⇒ Defining threads by implementing Runnable interface is a better approach when compare to extending Thread class for the following reasons:

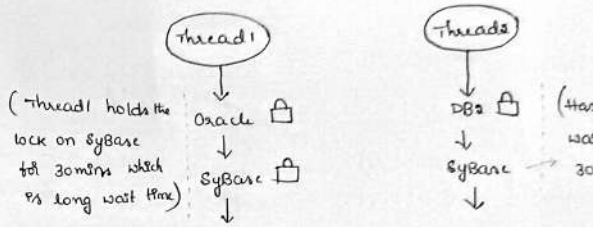
i) If an appln is created which doesnot require any methods of Thread class except run() & start() then extending Thread class can be avoided.

ii) If a class wants to have multithreading capabilities & also at the same time inherit the features of another class then we can extend the required class & implement Runnable interface.

```
class A extends class B implements Runnable{
    ==========
}
```

(8) write a note on Livelock. & write a note on Starvation.

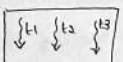
- If a thread has to wait for a locked resource for a time period more than expected then the thread is said to undergo Starvation. This is the result of a Livelock.
- A Livelock is long-extended wait time. It is not infinite.



(8) write a note on Race condition.

When we write a multithreaded pgm, the threads will always compete which each other for CPU time. As a result of this the order of execution of threads & hence the sequence of output will always change. This is called as Race condition.

In a pgm that expects output in a certain sequence Race condition must be avoided. It can be achieved by using join() method.



Without join():

```
t1.start();
t2.start();
t3.start();
```

o/p:1	o/p-2:	o/p-3:
t1	t2	t1
t3	t3	t2
t2	t1	t3

Race condition

With join():

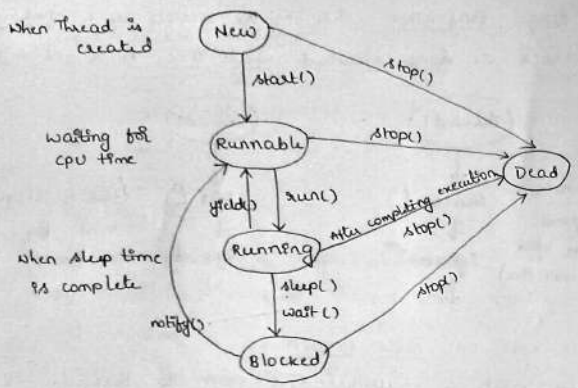
```
t1.start();
t1.join();
t3.start();
t3.join();
t2.start();
```

o/p-1:	o/p-2:	o/p-3:
t1	t1	t1
t3	t3	t3
t2	t2	t2

23/9/23

Thread life-cycle :

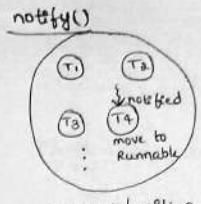
Different states of a thread :

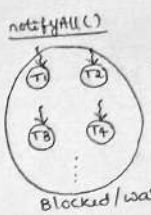


yield() : The process of moving a running thread to Runnable state to allow other waiting threads to execute is called yielding & can be achieved using yield().
If there are no waiting threads (or) if all waiting threads have lower priority than current thread then current thread will continue execution.

wait() & notify() : These 2 methods are used to achieve inter-thread communication. The thread which calls wait() method halts its execution & moves into block state until another thread calls notify() method.

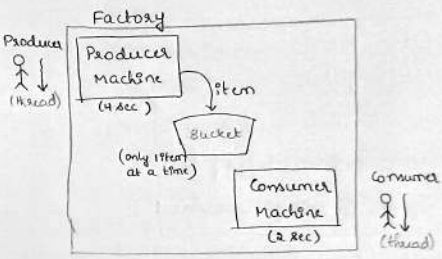
notify() v/s notifyAll() method :
When there are multiple waiting threads, if we use notify() method, any one of the threads will be notified whereas if we use notifyAll() method, all the waiting threads will be notified.





Here, all threads are notified & all threads move to Runnable.

Q. Explain Producer-consumer pblm & illustrate Inter-Thread communication.



Producer	consumer
I1 notify() → wait()	I1 notify() wait()
I2 notify() → wait()	I2 notify() wait
I3 notify() → wait()	I3 notify() wait()
I4 notify() → wait()	I4 notify() wait()
I5 notify() → wait()	I5 notify() wait() X → Deadlock

```
class Factory{
  int item;
  synchronized void produceMachine(){
      item++;
      S.o.pln("Item" + item + "is under production");
      try{
          Thread.sleep(4000);
          S.o.pln("Item" + item + "has been produced");
          notify();
          wait();
      }
      catch(Exception e){
          S.o.pln(e.getMessage());
      }
  }
  synchronized void consumeMachine(){
      S.o.pln("Item" + item + "is being consumed");
      try{
          Thread.sleep(2000);
          S.o.pln("Item" + item + "has been consumed");
          notify();
          if(item < 5){
              wait();
          }
      }
      catch(Exception e){
          S.o.pln(e.getMessage());
      }
  }
}
```

if we remove if block then it leads to deadlock

```
class Producer extends Thread{
    Factory x;
    Producer(Factory f){
        x = f;
    }
    public void run(){
        for(int i=1; i<=5; i++){
            x.produceMachine();
        }
    }
}
class Consumer extends Thread{
    Factory x;
    Consumer(Factory f){
        x = f;
    }
    public void run(){
        for(int i=1; i<=5; i++){
            x.consumeMachine();
        }
    }
}
class Demo{
    psvm(){
        Factory f = new Factory();
        Producer p = new Producer(f);
        Consumer c = new Consumer(f);
        p.start();
        c.start();
    }
}
```

24/9/23

Collection Framework :

Collection Framework was introduced to overcome the limitations of Arrays.

Limitations of Arrays are as follows :

i) Arrays are fixed in size. Hence they are also called as static arrays.

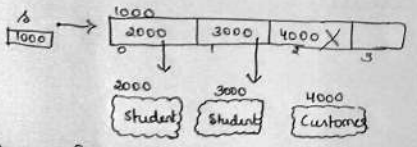
eg:
```
int a[] = new int[5];
a[0]=10;
a[1]=20;
a[2]=30;
a[3]=40;
a[4]=50;
X a[5]=60;  // AIOOBE.
```

ii) Only homogeneous elements can be stored within arrays.

eg:
```
int a[] = new int[5];
a[0] = 10;
a[1] = 20;
a[2] = 33.33f; X
a[3] = "Pentagon"; X
```

Note :- Arrays can be used to store both primitive data as well as objects (Non-primitive)

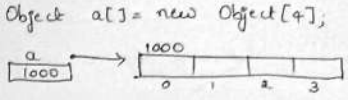
eg: Student s[] = new Student[4];




```
s[0]=new Student();
s[1]= new Student();
A[2] = new Customer; X
```

- There is a special case where arrays can store Heterogeneous elements i.e using Array of type Object.

eg: Object a[] = new Object[4];




```
a[0]=new Student();
a[1]=new Customer();
a[2]=new Employee();
```

iii) There is no underlying Data structure for static arrays. Hence there is no ready-made method support.

a.___() X

Collection :

A collection is a group of individual Objects represented as a single-entity.

Collection Framework :

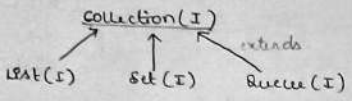
A set of classes & interfaces that enable us to represent a group of individual objects as a single entity is called collection Framework.

Differences b/w Array & Collection :

Array	Collection
• Arrays are fixed in size.	• Collections are growable in nature.
• Can hold Homogeneous elements.	• Can hold both Homogeneous & heterogeneous elements.
• No underlying Data structure hence no method support.	• Every collection has some underlying DS. hence there is readymade method support.
• Elements can be both primitive & objects (Non-primitive)	• Elements can only be objects.

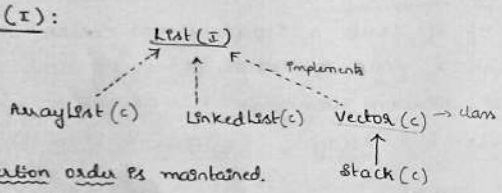
Interfaces of Collection Framework:

1. **Collection (I):**



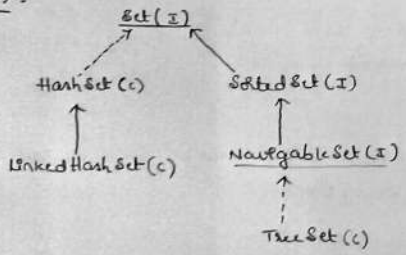
- Collection Interface is a Root interface of collection framework.
- It defines a set of methods that is implemented by very collection object.
- There is no concrete class that directly implements collection interface.

2. **List (I):**



- Insertion order is maintained.
- Duplicate elements are allowed.
- A list is an indexed collection of objects.

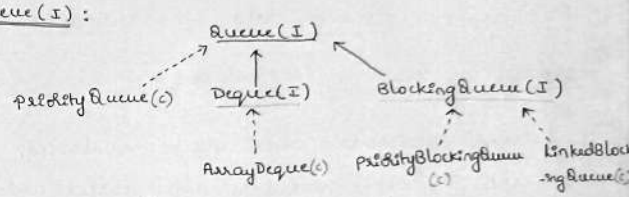
3. **Set (I):**



- Insertion order is not maintained.
- Duplicates not allowed.

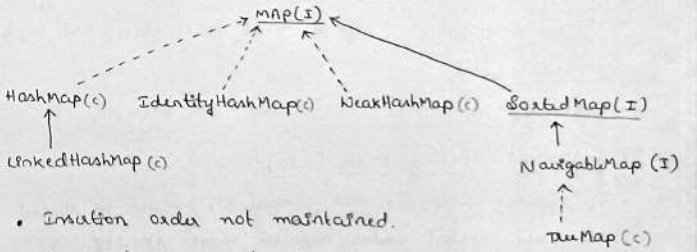
- We must go for SortedSet (I) if the collection must exhibit the following properties:
  - *) Elements stored in some sorted order.
  - *) Duplicates are allowed.
- The NavigableSet (I) defines a set of methods that help us to navigate the object that implements it.

4. **Queue (I):**



- Insertion & Deletion are in FIFO order.
- Duplicates are allowed.

5. **Map (I):**
- When every element of the collection must be a key-value pair where key is an object & value is also an object, we must go for Map(I).



- Insertion order not maintained.
- Duplicate keys not allowed.
- Duplicate values are allowed.

- **SortedMap(I)** : *) Elements are stored in some sorted order of keys.
  *) Duplicate keys are not allowed.
  *) Duplicate values are allowed.

- **NavigableMap(I)** : defines a set of methods that help us navigate the object that implements it.

Note: Map(I) is not a child of Collection(I).

Overview of Collection Framework:

Linked → Insertion order will be maintained.

Tree → Object stored in some sorted order.

Collections → [10, 20, 30, ...]

Map → {K1=V1 , K2=V2 .... }

* **Sorting order**:

Types:

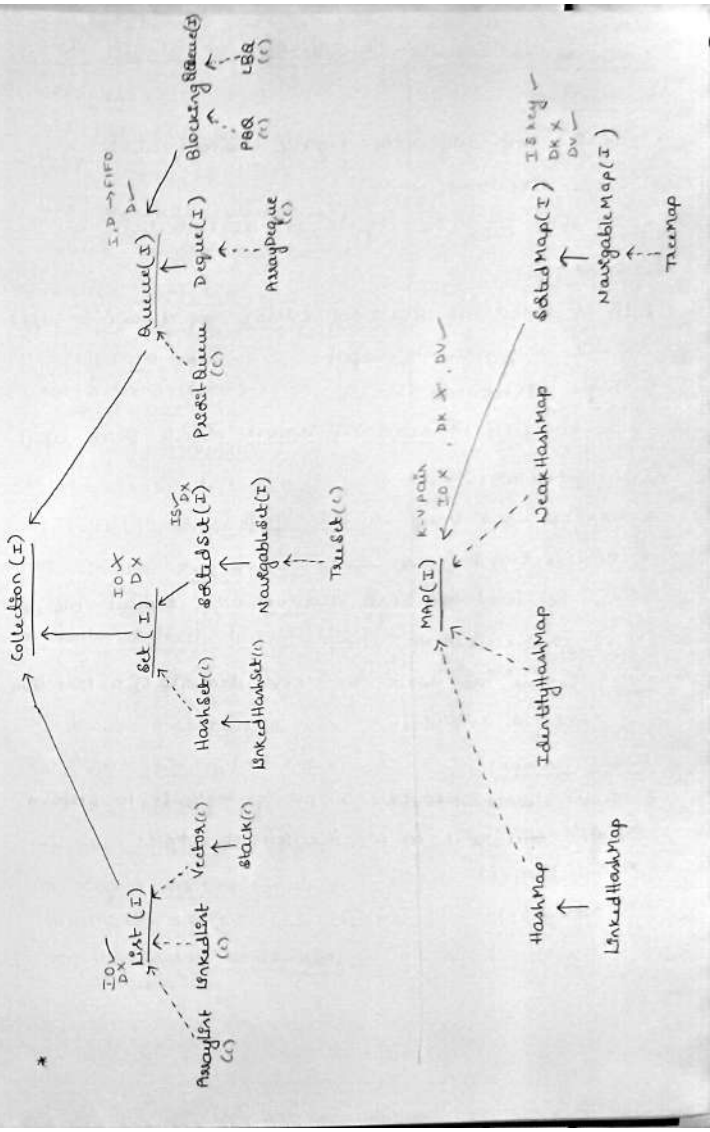
i) **Default Natural Sorting Order [DNSO]**:

- If objects are to be stored in DNSO then they must be comparable (objects must be of class that implements Comparable(I)).
- Internally compareTo() is called for DNSO.

ii) **Customized sorting Order [CSO]**:

- If objects which are not comparable must be stored in some sorted order then we must specify CSO.
- Internally compare() of Comparator(I) is called.

Note: DNSO is available only for comparable objects.



overview

* 3 cursors of collection framework:

i) Enumeration:

- can be used only with Legacy classes (Vector).
- Single directional (⟶)
- can only read the objects in the collection.

ii) Iterator:

- It is called as Universal cursor as it can be used with any collection object.
- Single directional (⟶).
- can be used to read & remove objects from collection.

3 methods of Iterator
hasNext(), next(), remove()

iii) ListIterator:

- can be used only on List objects.
- Bidirectional (⟶, ⟵)
- can be used to read, remove, add, replace objects in collection.

Note: Cursors are used to access elements of a collection one at a time.

* Utility classes:

These classes provide a set of methods to perform complex operations on a collection of objects

i) Collections (c)

ii) Arrays (c)

Methods defined in Collection(I):

i) void add(Object o)

ii) void addAll(Collection c)

iii) boolean/void remove(Object o)

iv) boolean/void removeAll(Collection c)

v) boolean retainAll(Collection c)

vi) boolean contains(object o)

vii) boolean containsAll(Collection c)

viii) int size()

ix) void clear()

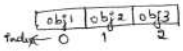
x) boolean isEmpty()

xi) boolean equals(Collection c) → compares both content & order

xii) Object[] toArray()

xiii) Iterator iterator()
(Interface)

Methods defined in List(I):



i) void add(int index, Object o)

ii) boolean addAll(int index, Collection c)

iii) Object remove(int index)

iv) int indexOf(Object o) → returns index value of passed no but it only returns first occurance of the given no.

v) int lastIndexOf(Object o)

vi) Object get(int index)

vii) Object set(int index, Object o)

viii) ListIterator listIterator()
(Interface)

ArrayList :

- Insertion order is maintained.
- Duplicates are allowed.
- Underlying DS is resizable (or) growable array.
- Both homogeneous & heterogeneous elements can be stored in ArrayList.
- null insertion is allowed.

Constructors of ArrayList :

i) ArrayList al = new ArrayList();

- By using above constructor we can create an Empty ArrayList with Default capacity = 10.
- The formula to update capacity is

New capacity = old capacity * 3/2 + 1.

ii) ArrayList al = new ArrayList (int capacity)

- By using above constructor we can create an ArrayList of any capacity.

iii) ArrayList al = new ArrayList (Collection c);

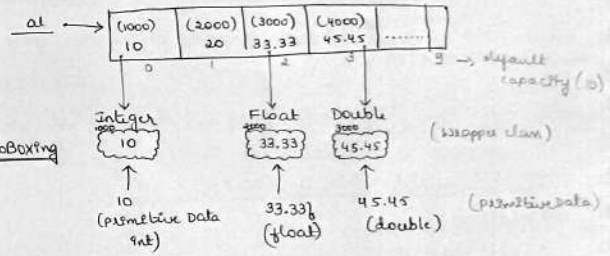
- The above constructor is used to convert any collection object into ArrayList.

AutoBoxing V/s UnBoxing :

- The process of converting primitive-data into its corresponding wrapper class objects is called AutoBoxing.
- This process is performed before placing data into the collection.
- The process of converting wrapper class objects into corresponding primitive data is called Unboxing.

Primitive Data type		corresponding Wrapper class
byte	-	Byte
short	-	Short
int	-	Integer
long	-	Long
double	-	Double
float	-	Float
char	-	Character
boolean	-	Boolean

Eg:
```
ArrayList al = new ArrayList();
al.add(10);        al.add(33.33f);
al.add(20);        al.add(45.45);
```



Generics in Java :

This concept is used to make a collection strictly Homogeneous. It is a combination of wrapper class & < > angular brackets.

```
ArrayList<Integer> al = new ArrayList<>();
```

Methods defined within Enumeration:

- boolean hasMoreElements()
- Object nextElement()

Method defined within Iterator:

- boolean hasNext()
- Object next()
- void remove()

Methods defined within ListIterator:

- boolean hasNext()
- Object null()
- int nextIndex()
- boolean hasPrevious()
- Object previous()
- int previousIndex()
- void add(Object)
- void remove
- void set(Object)

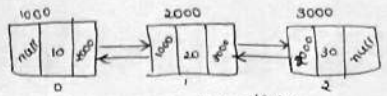
Methods defined within LinkedList:

- void addFirst()
- void addLast()
- Object getFirst()
- Object getLast()
- Object removeFirst()
- Object removeLast()

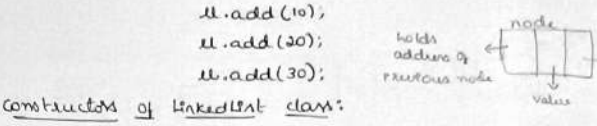
LinkedList:

- Insertion order is maintained.
- Duplicate elements are allowed.
- Underlying DS is Doubly Linked List.
- Both homogeneous & heterogeneous objects can be stored.
- null insertion is possible.

Visualization:




```
LinkedList ll = new LinkedList();
ll.add(10);
ll.add(20);
ll.add(30);
```



Constructors of LinkedList class:

i) `LinkedList ll = new LinkedList();`
Node is created only when element is added to LinkedList.

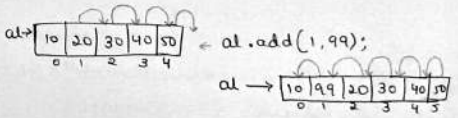
ii) `LinkedList ll = new LinkedList(Collection c);`

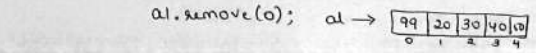
Q) How to choose b/w ArrayList & LinkedList?

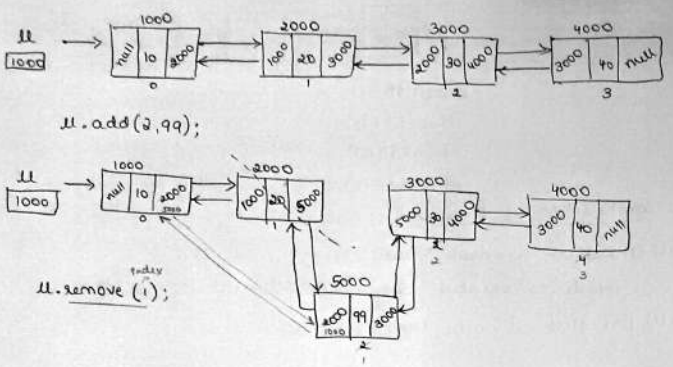
A)

- ArrayList & Vector are 2 classes of collection framework that implement RandomAccess interface. As a result of this time taken to access first element & last element (any element) is the same.
- In order to access a particular node of LinkedList we must always start from the first node & traverse all subsequent nodes until we reach the required node. Hence if the frequent operation of the pgm is retrieval, ArrayList is the best choice.
- If the frequent operation of the pgm is insertion (or) Deletion, LinkedList would be the best choice. as there is no shifting required.

eg:







u.add(2,99);

u.remove(1);

Vector class :

27/9/23

- It is a Legacy class that means it was included in Java 1.0 version
- All the properties of vector are same as that of ArrayList & Vector even implements RandomAccess Interface.
- *** The only diff b/w them is methods of ArrayList are not synchronized whereas methods of vector are synchronized.

Constructors of vector class:

i) Vector v = new Vector();
- Empty vector with Default capacity = 10
- New capacity = old capacity * 2.

ii) Vector v = new Vector(int capacity)

iii) Vector v = new Vector(int capacity, int incremental capacity)

iv) Vector v = new Vector(Collection c)

Methods of Vector:

- addElement(Object o)
- insertElementAt(int index, Object o)
- removeElement(Object o)
- removeAllElements(Collection c)
- removeElementAt(int index)
- firstElement()
- lastElement()
- Enumeration enumeration()

Stack class :

(FILO)

- It is used to achive First-In-Last-Out order for Insertion & Deletion

Properties: same as ArrayList.
- Underlying DS is stack.

constructors of stack class:

i) Stack s = new Stack()

ii) Stack s = new Stack(Collection c)

Methods of stack:

- Push(Object o)
- Object pop()
- Object peek() → to display top element on stack
- empty()
- int search(Object o) → -1 if element is not found
  → offset from top if element is found.

Set (I) :

No additional methods present in Set(I) (same as Collection(I).

HashSet :

- Insertion order is not maintained. but Elements are stored in according to their hash code.
- Duplicates are not allowed.
- Underlying DS is hash table.
- Both Homogeneous & heterogeneous objects allowed.
- null insertion is possible.
- The speed of searching operation will be significantly increased by using the concept of Hashing.
- (No additional methods defined in HashSet class.)

Constructors of HashSet :

i) HashSet hs = new HashSet(); (Default capacity=16)

ii) HashSet hs = new HashSet(int capacity)

* iii) HashSet hs = new HashSet(int Capacity, float FillRatio)

iv) HashSet hs = new HashSet(Collection c)

- Default capacity = 16 (we can update capacity in b/w collection)
- FillRatio (or) LoadFactor indicates the portion of current capacity that must be occupied in order to update it
  Updated capacity = old capacity * 2.
  - Default loadFactor = 0.75

LinkedHashSet :

- Insertion order is maintained.
- Duplicates are not allowed.
- Underlying DS is Hash table + Linked list.
- Both homogeneous & heterogeneous objects allowed.
- null insertion is possible.
- Constructors of LinkedHashSet → Refer to HashSet (names with change) (LinkedHashSet hs = new LinkedHashSet())
- Methods of LinkedHashSet → No additional methods.

Sorted Set

Methods specified in SortedSet(I):

- Object first()
- Object last()
- SortedSet headSet(Object o) → eg: [10, 20, 30, 40, 50] headSet(30) → [10, 20]
- SortedSet tailSet(Object o) tailSet(30) → [30, 40, 50]
- SortedSet subSet(Object o1, Object o2) → subSet(20, 40) → [20, 30]
- Comparator comparator()

Common for HashSet, LinkedHashSet, HashMap, LinkedHashMap, IdentityHashMap, Hashtable, WeakHashMap.

(Methods of NavigableSet(I) - No methods are present in NavigableSet(I) needs to be studied at this stage.)

TreeSet class:

- Insertion order is not maintained.
- Elements are stored according to some sorting order.
- Duplicates are not allowed.
- Underlying DS is balanced tree.
- Only Homogeneous objects can be stored. Heterogeneous objects are allowed only if customized sorting order is specified.
- null insertion allowed only once. as the first element.
- Methods defined within TreeSet : No additional methods.

Constructors of TreeSet :

i) TreeSet t = new TreeSet();
   - Elements stored as per Default Natural sorting order.

ii) TreeSet t = new TreeSet(Comparator c)
   - Elements stored as per customized sorting order.

iii) TreeSet t = new TreeSet(Collection c)

Note :

- DNSO is specified only for inbuilt classes but not all inbuilt classes.
- only those classes that implement Comparable(I) have DNSO
- Objects of classes that implement Comparable(I) are called comparable objects.
- When objects are stored a/c to DNSO, compareTo() method is called internally.

- CSO (customized sorting order) is used to store objects that are not comparable in the specified sorting order & also if you we are not happy with DNSO.
- If objects are stored a/c to CSO, compare() method of Comparator(I) is called internally.

5/10/23

Eg for DNSO :

Homogeneous (Integer generic)

```
TreeSet<Integer> ts = new TreeSet<>();
```

Add 20, 40, 30, 10, 50 to above TreeSet.

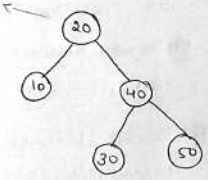
t.add(20); → start comparing from root node

t.add(40); → 40.compareTo(20) // ①

t.add(30); → 30.compareTo(20) // ①
30.compareTo(40) // -1

t.add(10) → 10.compareTo(20) // -1

t.add(50) → 50.compareTo(20) // ①
50.compareTo(40) // ①



[10, 20, 30, 40, 50]

Start collection from left most down node


```
=> interface Comparable {
       int compareTo(Object obj2);
   }
```

obj1 → Element to be added.
obj2 → Element already added.

Return values:

0 → Element already present (Duplicate)
+ve → Element must be added/placed to right side.
-ve → Element must be placed to left side.

In this eg, Elements are stored as per DNSO as Comparator type Object has not been passed to constructor of TreeSet.

Eg to create a TreeSet to store Strings in descending order of their length (CSO)

Note: equals() method of comparator(I) need not be implemented as it is provided by Object class.

```
=> interface Comparator {
       int compare(Object obj1, Object obj2);
       boolean equals(Object o);
   }
```

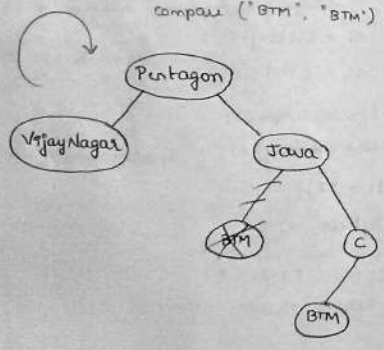
```
=> class MyComparator implements Comparator {
       public int compare(Object o1, Object o2) {
           String s1 = (String) o1;
           String s2 = (String) o2;        } Downcasting
           int l1 = s1.length();
           int l2 = s2.length();
           if (l1 > l2) {
               return -1;
           }
           else if (l1 < l2) {
               return 1;
           }
           else {
               if (s1.equals(s2)) return 0;
               else return 1;
           }
       }
   }
```

Class Demo1
psvm(){

TreeSet<String> t = new TreeSet<>(new MyComparator());
object of class MyComparator

*) Add "Pentagon", "Java", "C", "VijayNagar", "BTM", "BTM"

t.add("Pentagon");
t.add("Java"): → compare("Java", "Pentagon") // ①
t.add("C"); → compare("C", "Pentagon") // ①
compare("C", "Java") // ①
t.add("VijayNagar"): → compare("VijayNagar", "Pentagon") // -1
t.add("BTM"); → compare("BTM", "Pentagon") // ①
compare("BTM", "Java") // ①
compare("BTM", "C") // -1
t.add("BTM"): → compare("BTM", "Pentagon") // ①
compare("BTM", "Java") // ①
compare("BTM", "C") // -1
compare("BTM", "BTM") // ⓪
Sop(t);
}
}



[VijayNagar, Pentagon, Java, BTM, C]

6/10/23

- User defined classes can be given NSO by implementing Comparable(I) & overriding compareTo() method.

eg:
```
import java.util.*;
class Student implements Comparable{
    String name;
    int age;
    Student(String name, int age){
        this.name = name;
        this.age = aage;
    }
    public int compareTo(Object obj2){
        Student s2 = (Student) obj2;   Downcasting
        int a1 = this.age;   obj1
        int a2 = s2.age;   obj2
        if(a1 > a2){
            return 1;
        }
        else if(a1 < a2){
            return -1;
        }
        else{
            return 1;
        }
    }
    public String toString(){   If we don't implement toString() then address will be printed
        return name + " - " + age;
    }
}
class TreeSetStudentDemo{
    psvm(){
        TreeSet<Student> t = new TreeSet<>();
        t.add(new Student("Aagun", 27));
        t.add(new Student("Geetika", 22));
        t.add(new Student("Pragwal", 18));
        t.add(new Student("Suhas", 24));
        Sop(t);
    }
}
```

Queue(I):

(FIFO)

- Used to achive First-in-First-out order for insertion & Deletion.
- Duplicates are allowed.

Methods of Queue(I):

- offer(Object o)
- Object peek() — will give null if operation fails
- Object poll() — will give null if operation fails
- Object element() — will give exception if operation fails.
- object remove() — will give exception if operation fails.

PriorityQueue class:

methods → collections + Queue

Properties:

- Elements are stored in some priority order.
  Priority order can be DNSO / CSO.
- Duplicates are allowed.
- Elements must be homogeneous for DNSO & can be heterogeneous for CSO.

constructors of priorityQueue:

i) PriorityQueue pq = new PriorityQueue()
   - Default capacity = 11
   - DNSO

ii) PriorityQueue pq = new PriorityQueue(int initialCapacity)
   - DNSO

iii) PriorityQueue pq = new PriorityQueue(int initialCapacity, Comparator c)
   - CSO

iv) PriorityQueue pq = new PriorityQueue(Collection c)

Deque(I):

It stands for double-ended queue where elements can be added & removed from both the ends of the queue.

Methods of Deque(I):

- addFirst(Object o)
- addLast(Object o)
- offerFirst(Object o)
- offerLast(Object o)
- Object removeFirst()
- Object removeLast()
- Object pollFirst()
- Object pollLast()
- Object peekFirst()
- Object peekLast()

ArrayDeque class:

- Elements are stored in order of insertion.
- Duplicates are allowed.
- Both homogeneous & heterogeneous objects can be stored.

constructors of ArrayDeque:

i) ArrayDeque aq = new ArrayDeque();
   ↳ Default capacity = 16.

ii) ArrayDeque aq = new ArrayDeque(int initialCapacity)

iii) ArrayDeque aq = new ArrayDeque(Collection c)

BlockingQueue(I):

- This type of queue waits for the queue to become non-empty (or) non-full in order to complete add / remove operations. It will not instantly give null / an exception if the operation fails.
- In PriorityBlockingQueue elements are stored as per priority order. (DNSO / CSO).
- In LinkedBlockingQueue elements are stored in insertion order

Map(I):

- It is not a child of collection(I).

Methods of Map(I):

- put(Object key, Object value)
- putAll(map m)
- remove(Object key)
- Object (value) get(Object key)
- containsKey(Object o)
- clear()
- isEmpty()
- size()
- containsValue(Object o)
- ~~Sorted~~Set keySet()
- Collection values()
- ~~Sorted~~Set entrySet()

entry → key + value

To access elements one by one in map then (Iterator is not available in map) so we use entrySet. 7/10/23

HashMap class:

- Insertion order is not maintained. Elements are stored a/c HashCode of keys. (Random order)
- Duplicates → keys (X), values (✓)
- Heterogeneous → key (✓), value (✓)
- Underlying DS is hash table.

constructors of HashMap: refer to hash set.

Linked HashMap:

- Insertion order is maintained.
- Duplicates → keys (X), values (✓)
- Heterogenous → key (✓), value (✓)
- Underlying DS is hash table + LinkedList.
- constructors → refer HashSet.

SortedMap(I) methods:

- Object firstKey()
- Object lastKey()
- SortedMap headMap(Object key)
- SortedMap tailMap(Object key)
- SortedMap subMap(Object key1, Object key2)
- Comparator comparator()

TreeMap class:

- Insertion order is not maintained.
- Elements are stored in some sorting order of keys.
- Duplicates → keys (X), values (✓)
- Heterogenous → keys (X), values (✓)
- Underlying DS is red-black-tree.

interface Map{ = interface Entry{ = } }

constructors of TreeMap:

i) TreeMap t = new TreeMap(); → DNSO

ii) TreeMap t = new TreeMap(Comparator C);

iii) TreeMap t = new TreeMap(Map m)

Note: tm → 1 = Vishal, 2 = Raj, 3 = Ajay (no Iterator)

tm.elements()
tm.Iterator()
tm.ListIterator() } Error

```
Set sc = tm.entrySet();
sc => [1=Vishal, 2=Raj, 3=Ajay]
      (1 = key, Vishal = value; whole = entry)

Iterator it = sc.Iterator();
while(it.hasNext()){
    Map.Entry ele = (Map.Entry) it.next();
    S.o.println(ele.getKey());
}
```

1, 2, 3 → Integer class
Vishal, Raj, Ajay → String

1 = Vishal
2 = Raj
3 = Ajay
} map.entry.

⇒ Write a pgm to find frequency of all elements in a list.

- **Hashing:** Element → Information regarding element
  key → value
- **Eg:** al → [1, 1, 2, 1, 1, 3, 3, 2, 1, 2, 3, 4, 1, 1, 2, 2, 2]

Map:

K	V
1	7
2	6
3	3
4	1

- **Pgm:**

```
import java.util.*;
class HashingDemo{
    P S V m(){
        Scanner sc = new Scanner(System.in);
        ArrayList<Integer> al = new ArrayList<>();
        char choice = 'y';
        int val;
        while (choice == 'y'){
            S.o.println("Enter the element:");
            al.add(sc.nextInt());
            S.o.println("Press 'y' to enter one more element
                        and 'n' otherwise");
            choice = sc.next().charAt(0);
        }
        TreeMap<Integer, Integer> tm = new TreeMap<>();
        Iterator it = al.iterator();
        while (it.hasNext()){
            Integer ele = (Integer) it.next();
            if (tm.containsKey(ele)){
                val = tm.get(ele) + 1;
                tm.put(ele, val);
            }
            else{
                tm.put(ele, 1);
            }
        }
        S.o.println(tm);
    }
}
```

Methods of Collections Utility class:

i) addAll(Collection<T> c, T ele1, T ele2, ...); (T → type (generics))
ii) frequency(Collection c, Object o)
iii) min(Collection c)
iv) max(Collection c)
v) copy(List<T> source, List<T> Destination)
vi) reverse(List<T>)
vii) swap(List<T>, int i, int j);
viii) ~~sort~~ shuffle(List<T>) → DNSO
ix) sort(List<T>, Comparator c) → CSO

All methods are static.
↓
methods should be given along with class name
Collections
Collections.addAll(_)

Eg:

```
import java.util.*;
class IncLengthComparator implements Comparator<String>{
    public int compare(String s1, String s2){
        int l1 = s1.length();                    with <generics>
        int l2 = s2.length();
        if (l1 > l2){
            return 1;
        }
        else if(l1 < l2){
            return -1;
        }
        else{
            return 1;
        }
    }
}
class DecLengthComparator implements Comparator{
    public int compare(Object obj1, Object obj2){
        String s1 = (String)obj1;                without generics
        String s2 = (String)obj2;
        int l1 = s1.length();
        int l2 = s2.length();
        if (l1 > l2){
            return -1;
        }
        else if(l1 < l2){
            return 1;
        }
        else{
            return 1;
        }
    }
}
```

```
class UtilityDemo{
    p s v m(){
        ArrayList<String> al = new ArrayList<>();
        Collections.addAll(al, "Raj", "sharan", "Yathish", "Baggesh",
                               "Yathish");
        Collections.sort(al);
        S.o.println(al); // [B, R, S, Y, Y]
        Collections.sort(al, Collections.reverseOrder());
        S.o.println(al); // [Y, Y, S, R, B]
        Collections.sort(al, new IncLengthComparator());
        S.o.println(al); // [R, S, Y, Y, B]
        Collections.sort(al, new DecLengthComparator());
        S.o.println(al); // [B, Y, Y, S, R]
    }
}
```