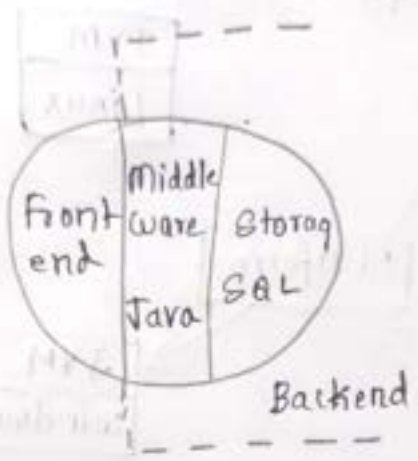
07/09/2023

☆ **Software:-**
It is a collection of programs used to perform a different task based on a business requirement.
In general, It is a automated solution for a real world problems

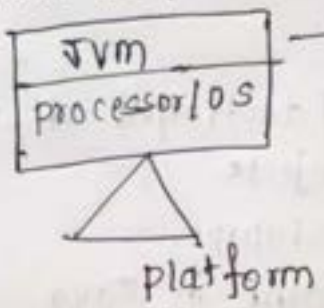


☆ **Programming:-**
It is a set of instruction given to a machine or platform to perform a specific task.

Programming Language:-
The language used to give set of instruction to a machine or platform.
Ex:- C, C++, Java, C#

The grammar part of programming language is called Syntax.

☆ **Platform:-** It is a place where we run all the applications and programs.
Platform is a combination of hardware and software.
i.e Hardware [processor] + Software [OS]



→ s/w only platform

Note:- Without any platform it is not possible to run any application.

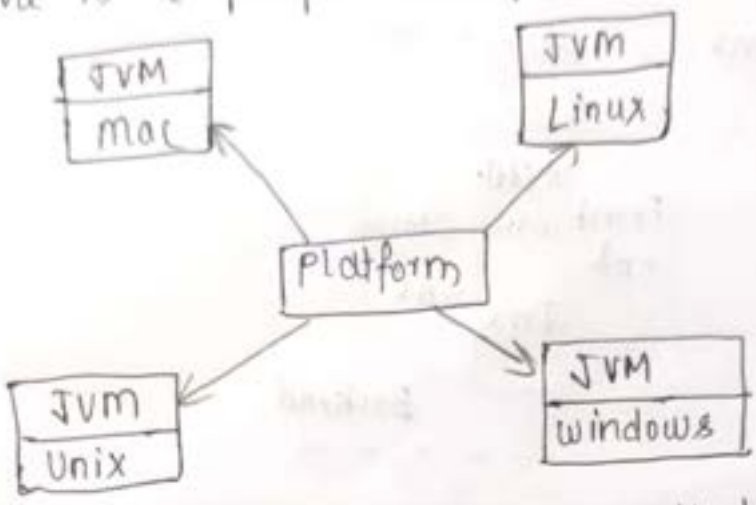
08/09/23

★ Java:-

- Java is a high-level object oriented programming language.

★ object oriented:- Pursuing everything in the form of object

- Java is a platform independent language



- Platform Independent:- The java application program developed in one platform can able to run on any other platform provided, JVM must be present.
- Java was developed in the year 1995 by James ~~Gosho~~ Gosling.
- Java file saved with .java extension.
- Java is pretty simple & has high performance.
- In java we need not to worry about memory allocation bcz itself has garbage collection.

★ Class:-

- class is a logical entity, it gets life only when the object is created.
- class is a blueprint/Master copy using which object is created.
- Without class it is not possible to create any object in java.
- When the object is created/destroyed it not affect the class not other similar objects.
- class in non-primitive datatype.
- Each objects are independent in Java.

☆ Object :-

- Object is a real world physical entity.
- Anything which is physically present which is considered as object.
- It is a instance of class.
- object creat with two things
  (i) States
  (ii) Behaviour.

(i) States :- It is a information / description about the object. It is technically called as Data member.

(ii) Behaviour :- It is a action performed by the object.
- It is technically called as method.

- class :- Human
  Object :- Human

States [Data Members]
- Name
- Age
- DOB

Behaviour [Methods]
- Eat()
- Sleep()
- Walk()

- class :- FlipKart
  object :- FlipKart

DataMember
- Customer name
- Phone num
- Mail id
- Password
- Address
- Pin
- Price
- Delivery date
- Bill address

Methods
- search()
- wishlist()
- Add_to_Cart()
- removeCard()
- buy()
- return()
- Cancel()
- sort()
- filter()

pointer works Internally

11/09/23

- Program
- Programming Language
- Platform, independent
- Class
- object

Class - Blueprint of Object

1) How to write a class?

```
Access_Modifier class_Name {
}
```

Ex:- class Student{ → Student.java

}

- To create any object its class must be created for
- The class name should be Noun.
- As per the coding standard the cls name first character should be written in Uppercase letter.
- If the cls name contain multiple words then we can follow PascalCase (i.e class PentagonSpace { })
-

2) Contents of class
- Data Members (State)
- Methods (Behaviour)
- Constructors
- Blocks
- Nested class

3) Keyword

In java there are some set of predefined words are present (Inbuilt) and each words is having predefined

meaning is called Keywords

Note*: Java is case sensitive. Hence all the Keywords must be declared with ~~case~~ lower case.

There are total 51 Keywords. In that 49 are in use and 2 are not in use(**)

Keywords

abstract	do	if	package	synchronized
boolean	double	implements	private	switch
break	else	protected	throw	this
byte	extends	instanceof	public	import
case	false	int	return	throws
catch	final	interface	short	transient
char	finally	long	static	true
class	float	native	strictfp	try
const**	for	new	super	void
continue	goto**	null	switch	volatile
default	assert			while

```
Class Employee
{ - - -
}
```
Error Keyword must be in lower case

```
Class EMPLOYEE
{ - - -
}
```
Here it will executed but we are not following Coding Standards

```
Class Employee
{ - - -
  - - -
  - - -
}
```
It follows Coding Standards

Ex:- class Employee

```
{
    emp_id = 400;
    name = "Pavithra";
    age = 25;
    Designation = "Software Developer";
    work();
    login();
    logout();
    takeleave();
    resign_Job();
}
```

State/ Data members

methods / Behaviour

4) Identifiers

Naming Components of Java members such as class, methods, State, constructors

Rules:-

- The Identifiers should not have whitespace.
- It should not start with number but can have no in name.
- It should not contain special character Except dollar ($) and underscore (_)
- It should not be Keyword.

```
class Pavithra 8K
{ ---
}
```

Invalid

Class

- Virat_18 = Valid name
- Pentagon_Space = Valid name
- 07_MSDhoni = Invalid name
- Pentagon@Btm = Invalid name

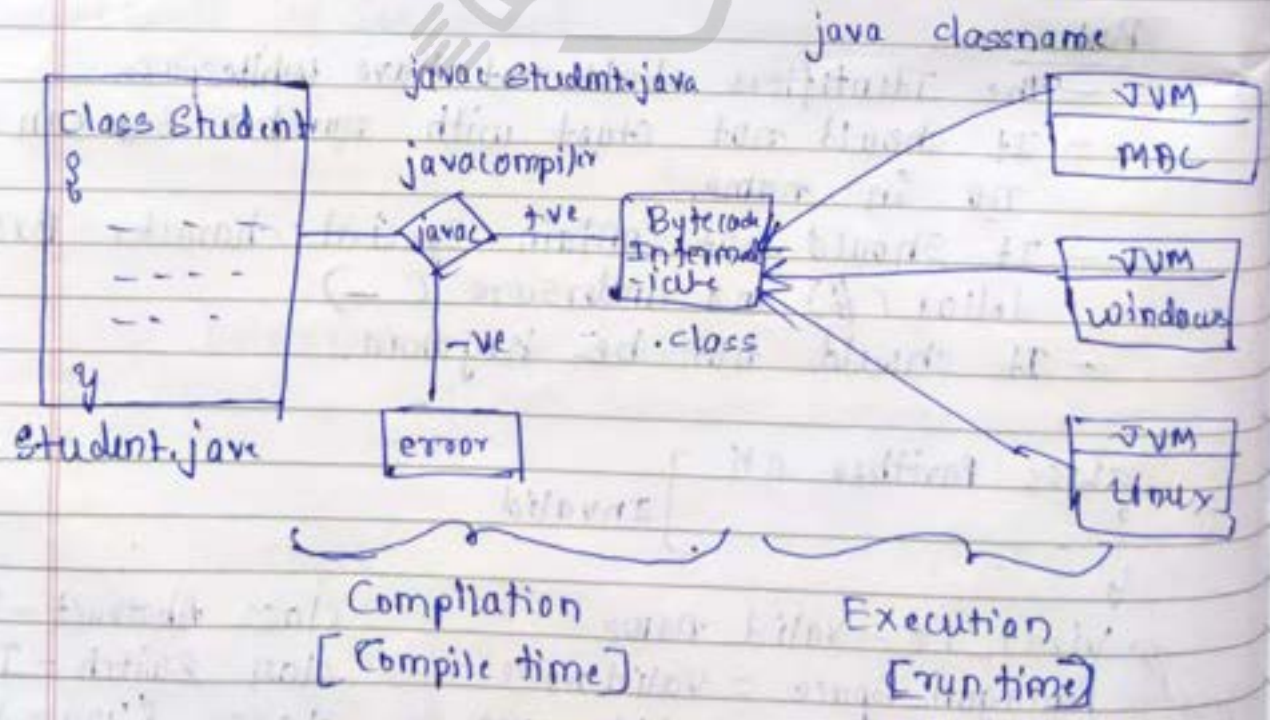
class Abstract - Valid
class Switch - Invalid
class _$420 - Valid

Software :-
A Set of programs used to perform the different task based on business requirement

12/09/2023

☆ Java Compilation & Execution (WORA)
WORA = Write Once Run Anywhere

Compilation :- It has 2 Steps
- First it checks the syntax, by using code
  javac filename.java.
- If the syntax is correct, then it generates the bytecode in form of classfile with .class extension. file will be saved in classname.class



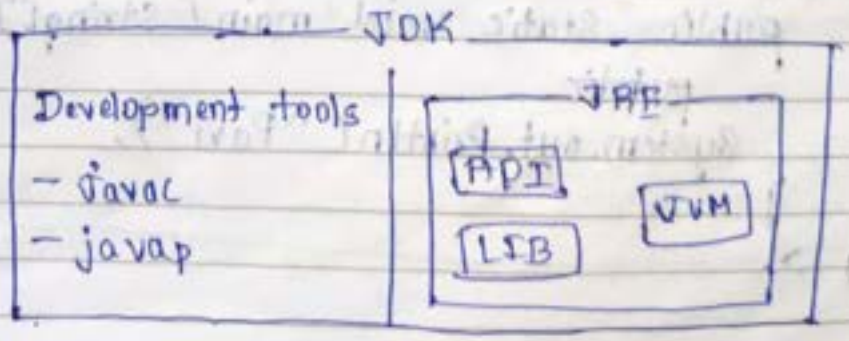
Byte Code :-

- It is neither high level nor a low-level Language
- It is Intermediate Language.
- It is platform independent
- It is generated by Compiler and interpreted by JVM
- It is stored in .class file

JVM :-

- Stands for Java Virtual Machine.
- It converts byte code into Machine level instruction or platform specific instructions.
- It is a platform dependent i.e windows jvm is different from mac jvm.
- It is a VM or platform (software only) that sits on actual platform (OS + Processor).
- It is an engine / environment for running/executing .class file.
- It is a set of instruction written in C and C++
- It is not a physical Component.
- It is responsible for memory allocation in java.
- Every OS will have its own JVM.
- ~~JV~~

JDK :-



- Stands for Java Development Kit.
- JDK is SDK (Software Development Kit)
- It contains necessary resources to compile & execute java based application programs.
- JDK internally contains JRE (Java Runtime Environment.
- latest version of JDK is java 20 which is released on March 2020.

JRE:-
- It is a runtime environment which is requir to execute java appli or byte code.
- It can be downloaded separately.
- Using JRE we can execute the code but we cannot develop the code.

Note:- To execute any program in Java the progra must contain main method.

Syntax:-

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

Scope of method

Example:-

```
class PrintName{ >
    public static void main (String[] args){
        ~~public~~
        System.out.Println("Pavi");
    }
} >
```

Scope of class

$2 \times 16 = 32$
2×16
32

Packages :- It includes programs of same type
Java files: Files which has java programs.

14-09-23

Datamembers :-
There are two types in data member
- Variable
- Constant

★ Variable :- It is a place where we store the data (03)
it is a name given to a memory location.
- The variable data can be varied (changed).
For ex :- Human age, weight, height etc.
Syntax :- Datatype variable_name [= data/value]

```
int age = 24;
double weight = 58.25;
float height = 5.7f
```

★ Constant :- Data which is fixed & it cannot be changed is called constant
Ex :- Human data-of-birth,
In mathematics value of PI etc
- In java constants are declared ~~where final~~ with 'final' keyword

Syntax :- final Datatype variable_name = data/value

★ Data :- Data is a raw fact which describes the attributes of an object.

Data

- Numeric Data
  - Decimal
  - Non-decimal
- Non-numeric Data
  - Alphabetical
  - AlphaNumeric
  - Special characters
  - AlphaNumericSpecial

```
class Human {
    age = "20";          // Numeric (non-decimal)
    name = "ding";       } Alphabetical
    gender = "male";
    height = 5.3;        } decimal
    weight = 56.8;
    dob = "02-Feb-1999";
    email = "dinga53@gmail.com"  } AlphaNumeric special
}
```

☆ Data Type :- In java or any other programming language data must be mentioned with its type. i.e. what type of data / information it is

- To specify the type of data we use data type
- Data types defines or specifies the type & size of the data. which is stored in memory location
- In java we have 2 different types of datatype
  - Primitive Datatype.
  - Non-primitive Datatype. (Reference / custom)

- Primitive datatypes
  - These are the basic inbuilt datatypes in java
  - It stores the raw data which holds the numeric values

- It also called as 'predefined' data type.
- It also allocate default value.

byte, int, short, long } Non decimal

double, float } decimal

boolean = True/False ⇒ 0/1

char = 1 character based on ASCII A = 65

Data Type	Size	range	default
byte	1 byte	-128 to 127 Imp	0
short	2 byte	-32,768 to 32,767	0
int	4 byte	-2,147,483,648 to 2,147,483,647	0
long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0
float	4 bytes	6 to 7 decimal	0.0f
double	8 bytes	15 decimal digits	0.0
boolean	1 bit	true/false values	false
char	2 bytes	store single character	\u000

Note - If we don't provide any values or data to the variables then by default it takes 'default value' based on the data type

- Non-primitive datatype
  - In java it is possible to declare or define our own type of data by using class and interfaces is called non-primitive datatype.

Ex:- Student, Employee, Staff, Trainer etc
- In Java there are few pre-defined non-primitive datatypes are presents.
Ex:- String, Array, Collection etc

Note:** Default value of all non-primitive datatypes are null.

String - It is an inbuilt class in Java, hence it is considered as non-primitive data type
- It is used to store sequence of charaters
- Any characters (AlphaNumeric, Special, Alphabets) which is stored by Double quotes are considered as String data

```
class Human
{
    String name = "Pavi";
    int age = 20;
    float height = 5.2f;
    double weight = 50.3;
    String email = "pavithrash@gmail.com";
}
```

-09-2023

☆ Object Creation
In java object can be created in different ways, in that the most common way of creating object is 'new' Keyword using.

There are 5 different ways to create an object
(1) Using new Keyword.

(ii) class.NewInstance()
(iii) clone
(iv) newInstance method of Constructor Class
(v) de-seridlization

(i) Object creation using 'new' keyword

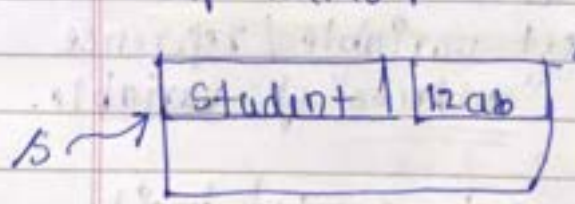
```
class Student{

    public static void main (String[] args)
    {
        Student s = new ~~Constructor();~~ Student();
        Datatyde  variable  Keyword  ~~Data value~~  Constructor

        System.out.println(s);
        Student s1 = new Student();
        S.O.p(s1);
    }
}
```

** The object is created by JVM in runtime inside the heap memory, when we used new keyword with construction
- All the objects are created at runtime.

Heap memory

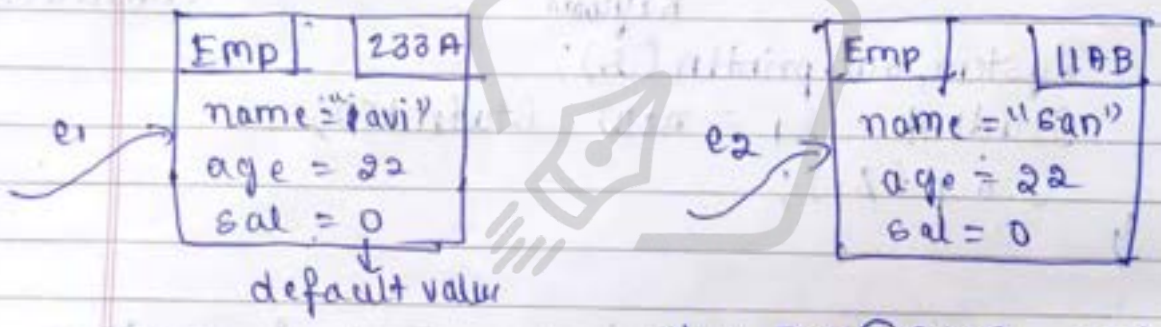


It prints the address of the variable.

for s, o/p = Student@12ab

```
Class Emp
{
    String name = "Pavi";
    int age = 22;
    long sal;
    public static void main(String[] args) {
        Emp e1 = new Emp(); // Object Creation
        Emp e2 = new Emp(); // Object Creation
        e2.name = "San";    changing value
        S.O.P(e1); S.O.P(e1.name);
    }
}
```

In heap memory



O/P:- Emp@233A, Pavi

Note:- • In java we can compile empty class we can execute empty class.

• To execute any program in java we have to write main method because ~~always~~ the execution always starts from main method.

• Whatever we print object variable / reference variable it always prints "address" of variable.

• An object of one class can be created inside another class.

```
class Pen {
}
```

```
class Mobile
{
}
Class Test
{
    public static void main (String[] args)
    {
        Pen P1 = new Pen();
        Mobile m1 = new Mobile();
    }
}
```

• Object is always represent with "hexadecimal format"

21/09/2023

Method

★ **Method:**

It is a set of instruction or block of code which is used to perform a specific task.

Syntax:-

```
Access_Modifiers return_type method_name ( datatype
var_name1, datatype varname2, ...,datatype varnamen)
{

    return value;
}
```

output

parameter (input to method)

return_type name parameter

Ex:- • public double mul (int a, double d)

```
{
    double product = a*d;
    return product;   → return statement
}
```

• public int add (int a, int b) {

```
    int sum = a+b;
    return sum;
}
```

* Rules to define method
  - Void method doesnot return any value.
  - Always method return type and the return value must have same datatype
  - Method can have only one return type.
  - A method can have only one return statement.
  - Return statement must be last executable statement inside a method
  - By using return statement we can return only one value but by using collection and array we can return multiple values.

Ex:- • void m()
```
{
    return 40;
}
```
→ O/P:- Error; bcz void method doesnot return any value

• void m()
```
{
    return;
}
```
→ No error Executed

• int m()
```
{
    return 420;
    S.o.p ("hello");
}
```
→ O/P:- Error; According to rule 5

- double m()

```
{
  return 55.25;
  return 40.25;
}
```

→ Error, bcz its not satisfy the 6th rule.

- String m()

```
{
  return "hello", "hi";
}
```

- String m()

```
{
  String name = "Pentagon Space";
  return name;
}
```

- String m()

```
{
  String name = "Pavi";
  return name+"SK";
}
```

No error

- In java, the methods are executed in runtime stack memory.
- For each method separate thread will be created and it will be executed in runtime stack.
- The method will never get executed unless until we invoke the method (calling the method).

Ex:- A Dancer (obj) dance (method) and throws a Jacket (obj)

```
class Jacket {
}
class Dancer {
    Jacket dance()
    {
```

```
        Jacket j = new Jacket();
        return j;
    }
    public static void main (String[] args)
    {
        Dancer d = new Dancer();
        d.dance();
    }
}
```

Ex: Cow gives milk
obj method obj

```
class Milk {
}
class Cow {
    Milk gives()
    {
        Milk m = new Milk();
        return m;
    }
    public static void main (String[] args)
    {
        Cow c = new Cow();
        c.gives();
    }
}
```

Ex: ATM dispense Money taking pin as a input
obj method obj ~~method~~

```
class Money {
}
```

```
class Pin{
}
class ATM{
    Money dispense(int pin)
    {
        Money m = new Money();
        return pint m;
    }
    public static void main(String[] args){
        ATM a = new ATM();
        a.dispense(1234);
    }
}
```

Argument

Ex:- Shopkeeper sells a product
obj method obj

```
class product
{
}
class shopkeeper{
    Product sells()
    {
        Product p = new Product();
        return p();
    }
    public static void main(String[] args)
    {
        Shopkeeper s = new Shopkeeper();
        s.sells();
    }
}
```

* Parameter :- It is an input to the method.
- Parameter is the one which holds the argument
- Argument :- It is a data/value which is passed to the method.

2-09-2023

Ex:- For method of Student

```
class Student{
    String name;
    int rollno;
    String Branch;
    void study()
    {
        S.O.P (name + " is Studying");
    }
    public static void main(String[] arg){
        Student s1 = new Student();
        s1.name = "pavi";
        s2.rollno = 22;
        s2.Branch = "CSE";
        S.O.P ( s1.name + " rollno=" + s1.rollno + " branch=
                + s1.Branch);
        s1.study();
        student s2 = new Student();
        S.O.P ( s2.name + " rollno=" + s2.rollno + " branch=
        + s2.Branch);
        s2.study();
    }
```

O/p:

```
pavi  rollno = 22   branch = CSE
pavi  is studying
null  rollno = 0    branch = null
null  is studying
```

Ex:- Performing Addition & Subtraction ~~in~~ cal.

```
class Calcalator {
    String Brand = "casio";
    String colour = "Black";   } Instance variable
    int add ()
    {
        int a = 10;
        int b = 20;   } Local variable
        int sum = a+b;
        S.O.P ("Sum of a and b : " + sum);
    }   return sum;
    int sub()
    {
        int a = 30;
        int b = 10;
        int dif = a-b;
        S.O.P ("Difference of " + a + " " + b + " : " + dif);
    }   return dif;
    public static void main(String[] arg) {
        Calculator c = new Calculator();
        c.add();
        c.sub();
    }
}
```

Ex:- class calculator { // By passing parameter.

```
    int add(int a, int b)
    {
        int sum = a+b;
        S.O.P ("The sum of a & b : " + sum);
        return sum;
    }
```

```
int sub(int a, int b)
{
    int result = a-b
    S.O.P("Difference of a & b :" + result);
}
public static void main (String[] args)
{
    Calculator c = new Calculator();
    c.add(2,3);
    c.sub(10,5);
}
}
```

EX: By taking result of addition method as input to sub() method

```
class Calculator{
  int add( int a, int b){
      int sum = a+b;
      S.o.p("addition of" + a + "&" + b + " =" + sum)
      return sum;
  }
  int sub( int a, int b){
      int total = a-b;
      S.o.p("sub of" + a + "&" + b + "=" + total);
      return total;
  }
  public static void main(String[] args){
      Calculator c = new Calculator();
      int val1 = c.add(100, 200);
      int val2 = c.sub(val1, 100);
  }
}
```

25-09-2023

Program Execution

Ex.a:-

```
1. class Car{
2. {
3.     String brand = "Hyndai";
4.     double milage = 14.5;
5.     void start()
6.     {
7.         S.O.P("Car starts");
8.     }
9.     void moves()
10.    {
11.        S.O.P("Car Moves");
12.    }
13.
```

```
13. public static void main (
    String [] args)
14. {
15.     Car c = new Car();
16.     c.start();
17.     c.changeGear();
18.     c.move();
19. }
20. void changeGear()
21. {
22.     S.O.P("Car gear changed
23. }
24. }
```

13. JVM starts the execution
14. Start of the method
15. Object Creation

Car	11AB
brand = "Hyndai"	
milage = 14.5	

→ In heap memory

16. calling start method, c.start() → 5. Method execution Start [runtime stack]
    6. Start of start method
    7. prints "car starts"
    8. End of start method

17. changeGear() method → 20. Method execution starts
    21. Start of method
    22. prints "car gear changed"
    23. End of method

18. C.moves() method called → 9. method execution start
10. start of moves method
11. prints "car moves" statement
12. end of moves method

19. The program execution ends

Ex. b

```
1. class Car
2. {
3. String brand = "Hyndai";
4. double milage = 14.5;
5. void start()
6. {
7.   S.O.P ("Car Start");
8. }
9. void move()
10. {
11.   S.O.P("Car moves");
12. }
13. }
```

```
1. class TestCar
2. {
3. ⇒ p s v m (String[] arg
4. {
5. Car c = new Car();
6.   c.Start();
7.   c.move();
8. }
9. }
```

Program Execution

3. JVM start the execution in TestCar class
4. {
5. Object creation

Car	11+B
brand = "Hyndai"	
milage = 14.5	

← Heap memory

6. c.start() → 5. Method execution start (car class)
6. {
7. Prints "car start" statement
8. }

7. c.move() ⟶ 9. method execution start (car class)
10. { – Start
11. prints "car moves" statement
12. } – end

8. The program execution ends

To execute: javac Car.java
javac TestCar.java
java TestCar

★ this [Keyword] :-
- It is used to access the states and behaviour of current invoking object.
- this keyword indicates current invoking object.
- this keyword can be used only inside non-static methods, blocks and also inside a constructor.
- this keyword cannot be used inside a static method not even inside main method. (coz main method is also a static method).

Ex:- class Horse
{
String name; or String name = "xxx";
void run()
{
S.O.P (this.name + "Horse running");
}
h1.name (accessing state)
void jump()
{
S.O.P (this.name + "Horse jumping");
this.walk();
this = h1 (represent current invoking object)
⟶ accessing method

```
void walk()
{
    S.o.p(this.name+"horse walking");

}
public static void main(String[] args)
{
    Horse h1 = new Horse();
    h1.name = "Brego";
    Horse h2 = new Horse();
    h2.name = "Chethak";
    h1.run();
    h2.jump();
    h2.run();
    h2.walk()
    h2.jump();
}
}
```

o/p:- javac Horse.java
java Horse

Brego Horse running	Chetak Horse running
Brego Horse jumping	chetak Horse walking
-||- horse walking	chetak Horse jumping
	chethak Horse walking

S.o.P(this); is same as S.o.P(h1); Hence it represents current invoking object (object which is invoke on method)

.09/2023

★ Variable

A place where we store the data or value

Types of variable — Variables

- Static variable
- Non-static variable → Instance variable / Local variable

(1) Local variable :-

- A variable which is declared inside a method, constructor or block is called local variable.

Ex:
```
class Demo{
    void m(){
        int i=10;
    }
    Demo(){
        double d=26.60;
    }
    {
        String s="pavi";
    }
}
```
method — local variable

constructor

Block

- Local variable can only be accessed within the scope, but not outside the scope.

Ex:
```
class Test{
    void m()
    {
        int a=10;
        S.O.p(a);
    }
    void m1()
    {
        S.O.P(a);
    }
}
```
class scope

Local scope or method scope

→ Not possible

- local variable cannot be static

- Local variables cannot be accessed by using object reference followed by dot (.) operator. and not even by using this keyword.
- Local variables must be initialized before we use
Imp** - Local variables doesnot have any default value.

Ex:- class Dog()
{
void sleep(){
int double hours;
S.O.P("hours); → Error
}
}

- Local variables gets created or stored inside a "stack memory" and it has limited lifespan i.e it gets created when the method execution get started and it gets deleted when the method execution ends
- Two local variables within the same scope cannot have same name, but 2 local variables within the different scope can have same name.

Ex:- class Test
{
void m1 (){
int a = 10; ✓
double a = 10.23; ✗
}
void m2(){
int a = 11; ✓
}
}

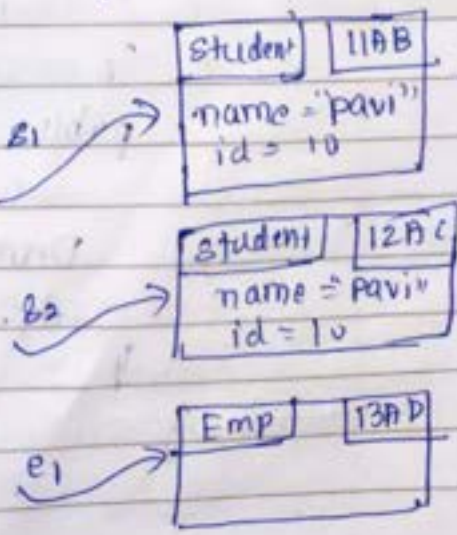
(ii) Instance variable:-
- An Instance variable is a non-static variable

which is declared inside a class but outside a method, constructor or block.

Ex: class Student {
    String name = "Pavi";
    int rollno = 098;] Instance variable
    void study()
    {
Method [double noh = 2.50; ⟶ Local variable
    }
    Student()
Constructor [{
    }
Block [{
    }
}

- Instance variable gets created only when the object is created.
- Instance variables gets created inside an object as a part of "heap memory."
- The no of copies of instance variable depends on no of object created (i.e if we create 30 objects then 30 times instance variable gets created.

Ex: class student {
    String name = "Pavi";
    int id = 10;
    p s v m (S[] arg) {
    Student S_1 = new Student();
    Student S_2 = new Student();
    Emp E_1 = new Emp();
    } }
    class Emp {
    }



- An Instance variable get initialized to default value based on data type (i.e if the user doesnot provide any value then it takes default value)
- An Instance variable life span is same as a object; i.e it gets created when the object get created. & it gets destroyed when the object get destroyed.
- The Scope of instance variables are throught the class and also outside the class.
- An instance variables can directly access inside a non-static method, block, constructor.
- An instance variables cannot be accessed directly inside a static method / block, but we can access by creating an object & by using object reference followed by dot operator.
- An instance variable can also be accessed outside the class by creating a object.

Ex:-
```
class Demo{
    int a = 10;
    void m(){
        S.O.P(a); ✓
    }
    Demo(){
        S.O.P(a); ✓
    }
    public static void main(){
        S.O.P(a); ✗
        Demo d1 = new Demo();
        d1.a = 20; ✓
    }
}
```

☆ Static Variables:-

- An variable which is declared with static keyword is called static variable.
- Static variables belongs to class but not to object.
- Static variables are also called as class variable
- Static variables can be accessed directly inside a static method without creating any object.

Ex:
```
class Student {
    int rollno;
    String name;
    Static int count;
    public static void main (String[] arg){
        Student s1 = new Student();
        s1.name = "Pavi";
        s1.rollno = 10;
        count = 5;
        s1.count = 11;
    }
}
```

- Static variables can accessed within the class and outside the class.
- Static variables can be accessed outside the class and also inside a non-static method by using class name followed by dot(.) operator.

Ex:
```
class Product {
    String pname;
    double price;
    static int no_of_products = 0;
    void m1() {
        Product.no_of_products = Product.no_of_product + 1;
        (or) Product.no_of_products++;
    }
```

```
class TestProduct {
    PSVM (String[] arg) {
        Product P1 = new Product();
            P1.pname = "mandi";
            P1.price = 1000.00;
            P1.m1();
            S.o.PC Product.no_of_products);
    }
}
```

- Static variables get loaded at the time of "class loading".
- Static variables get created inside a method area or class area.

Write a Difference b/w local variable & Instance variable
Write a Difference b/w Instance & Static variable

Local Variable	Instance variable
- Local variable is a variable which is declared inside a method, constructor, blocks.	- A variable which is declared inside a class but outside the method, constructor & blocks.
- These are accesed within the scope but not outside the scope	- These are accessed within the class and also outside the class.
- Variables cannot be accessed by object reference followed by dot operator & not even using this keyword.	- These can be accessed by object reference followed by dot operator inside a static method.
- It is stored in stack memory	- It is created & stored in heap memory.

- It doesnot have default value, it must be initialized before we use - It has limited lifespan, it gets created when the method execution start & it gets destroyed when the method execution end.	- It does have default value based on the data type. - It has lifespan same as the object, it gets created when object created & it gets destroyed when the object get destroyed.

Instance Variable	Static variable
- It is a ~~sto~~ non-static variable	- It is a static variable.
- It is declared inside a class & outside a method, constructor & block	- It is declared with static Keyword.
- It is created inside a heap memory.	- It is created inside a class memory area / method area
- It belongs to object	- It belongs to class.
- It can be accessed inside a ~~static~~ method by creating object	- Without creating object it can be directly inside a static method.
- It can be directly accessed within the non-static method	- It cannot be directly accessed inside a non-static method.
- These variables are accessed outside the class by creating object & by using object reference followed by dot operator.	- These variables are accessed outside the class & inside the non-static method by using classname followed by dot operator.

18-09-2023

Method Overloading

- In a class if we have more than one method with the same name but change in signature is called method overloading.
- Change in signature means
  (i) There should be change in no of parameters
  (ii) There should be change in datatype of parameter
  (iii) There should be change in sequence of parameter

Ex:-
```
class Test
{
    void add ()
    {
    }
    void add(int a)
    {
    }
    void add (int a, double b)
    {
    }
    double add (double b, int a)
    {
        return a+b;
    }
}
```

- Advantages:
  (1) method overloading helps to achieve "compile time polymorphism"

- Example for method duplication

```
Class Test {
  void m()
  {
  }
  int m()
  {
   return 10;
  }
}
```

method duplication ⟺

```
class Test {
  int m(double b)
  {
    return 10;
  }
  String m(double d)
  {
    return "hi";
  }
}
```

- ```
  class Test {
    void m(double d, String s)
    {
    }
    int m(double s, String d)
    {
      return 120;
    }
  }
  ```

  → Method duplication bcz it has same no of parameters, sequence and datatype

- ```
  class Test {
    void m1(int a)
    {
    }
    void m2(int b)
    {
    }
  }
  ```

  → It compile but neither method overloading nor duplication.

- Method overloading is used to perform same task by passing different parameters.
- Real time example for method overloading is "print() method"

Ex:- class FBwebsite

```
{
  void login( String uname, String pwd)
  {
    S.O.p("logged in using username & password");
  }
  void login (long contact, String pwd)
  {
    S.O.p ("logged in using contact & password");
  }
  void login (long contact, int Otp)
  {
    S.o.p("logged in using contact & otp");
  }
  public static void main (String[] arg){
      FBwebsite f = new FBwebsite();
      f.login("pavi", "pavi123");
      f.login("9353723547l", "pavi123");
      f.login(9353723547l, 0134);
  }
}
```

Ex:- class Test

```
{
  void m( int a)
  {
    S.o.p("I am int method");
  }
  void m(long a)
  {
    S.o.p ("I am a long method");
  }
```

```
public Static void main (String[] arg)
{
    Test t1 = new Test();
    t1.m(0);
}
```

0 is default value. it will call integer data type as first priority

Ex:-
```
Class Test {
  void m (String s)
  {
    S.O.P("String is printed");
  }
  void m (Student s)
  {
    S.O.P("Student type is executed");
  }
  public Static void main (String[] arg)
  {
    Test t1 = new Test();
    t1.m(null);
  }
}
```

$\longrightarrow$ It throws a error because all non-primitive datatypes has default value "null".

**- "println & print" method is an realtime example for method overloading.

There are 11 overloaded print & println methods are present

(i) public void println();
(ii) public void println(int n);
(iii) public void println(double d);
(iv) public void println(boolean b);
(char c);

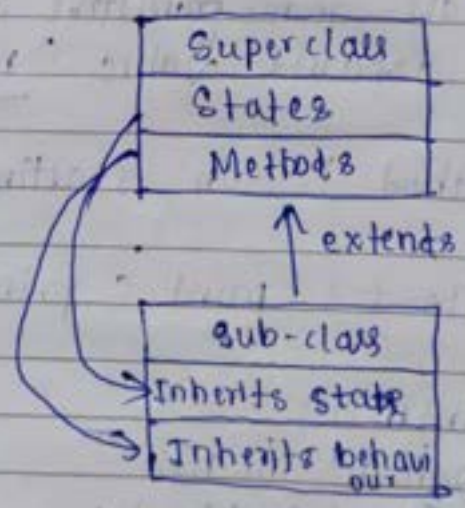
When a class represent the shared characteristics of more than one class is generalization Ex: vehicle is generalization of bike, car, truck

(float f);
(long l);
(char[] c);
(String s);
(object o);
(byte b);

Note:- Static methods can be "overloaded" even main method can be overloaded.
In this case JVM will always execute main method with states "String[] arg"

03/10/2023

★ Inheritance (Imp★★)
- The process of acquiring the properties of one class (Parent / Super class) to another class (child / Sub class) is Inheritance
- We can achieve Inheritance by using extends Keyword
- It is a top-down approach.



Ex: class Parent
{

IS-A relationship. A class inherited from another class can be used as superclass.

```
    void m1() {
    }
}
class child extends Parent {
    void m2() {
    }
}
```

→ Subclass Specific method

- Class which has a common properties are called Superclass | Parent class | Base class.
- The class which inherits the properties from the Superclass are called Sub class | Child | Derived class.
- Inheritance is also known as "Is-A" relationship.
- Advantages :-
  - We can avoid code redundancy | duplication.
  - We can achieve code reusability.
  - We can achieve generalization
  - Inheritance indirectly helps to achieve "polymorphism"

Ex:-
```
class vegetable{    // Super class
    double price;
    double qty;
    String name;
  void wash()
  {
    S.O.p ("Washing a vegetable");
  }
  void chop()
  {
    S.O.p ("Vegetable chopped");
  }
}
```

```
class Carrot extends Vegetable // Sub class
{
    void prepareHalwa() // Subclass specific method
    {
        S.O.p("Halwa prepared");
    }
}
class Chilly extends Vegetable // Sub class
{
    void prepareBajji() // Subclass specific method
    {
        S.O.p("Bajji Prepared");
    }
}
class TestClass {
    public static void main (String[] args)
    {
        Vegetable v = new Vegetable();
        v.price = 100;
        v.qty = 10;
        v.wash();
        v.prepareHalwa(); X  -> Can't access subclass specific var/method
        Carrot c = new Carrot();
        c.price = 110;
        c.qty = 12;, c.name "Sweet Carrot";  -> accessing inherited properties
        c.wash();
        c.chop();  -> Accessing inherited method
        c.prepareHalwa();  -> Accessing subclass specific meth
    }
}
```

Note:- By using subclassObject Reference followed by dot operator we can access

By creating object we can access the method & variable of the final class (String, Scanner, System, Integer, StringBuffer, StringBuilder) etc

- Inherited variably.
- Inherited methods.
- Subclass specific variables and methods.

• By using Superclass Object we can access inherited variables and methods but we cannot subclass specific variables & methods.

Imp** • By using Inheritance we can inherit variables and methods but not constructor & blocks.

• Final class cannot be extended i.e final class cannot have subclass.

Ex:- final class Human
{
}
class Emp extends Human
{
}

final class cannot have subclass

(class, method & variable can be constant).

- Private variables and methods cannot be inherited

Ex:- class Father
{
double money = 1200000;
private String gfName = "Alia bhat";
void walk()
{
}
private void smoke()
{
}
}
class Son extends Father
{

```
public static void main () {
    Son s = new Son()
    s.money = 2000
    s.op (s.money);
    S.o.p (s.gfName); X ← private variable
    s.walk();
    s.smoke(); X // private method
  }
}
```

Imp** - Static variable & methods cannot be inherited into subclass.

14/10/2023

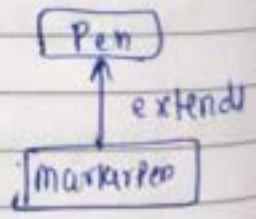
★ Types / Levels of Inheritance

(i) Single level Inheritance
(ii) Multi level Inheritance
(iii) Hierarchical Inheritance
(iv) Multiple Inheritance
(v) Hybrid Inheritance

(1) Single level Inheritance :

- A class extends to another class i.e one super class & one subclass is single-level-inheritance

Ex:-
```
class Pen {
}
class MarkerPen extends Pen {
}
```



Note:- In Java each & every class by default automatically extends to Supermost class Object called "Object"

Imp • Object

- Object class is a root class for all the classes in Java.
- Object class is present in java.lang package

Methods of object class

- protected Object clone()
  public boolean equals(Object obj)
  protected void finalize() throws Throwable
  public final Class getClass()
  public int hashCode()
  public String toString()
  public final void notify()
  public final void notifyAll()
  public final void wait()
  public final void wait(long timeout)
  public final void wait(long timout, int nanos)

Ex: • class Emp { extends → object
}

• class Demo { → Object
}

class Pen { → Object
}
class SketchPen extends Pen {
}

Note: If a Subclass extends to another superclass then it doesnot inherit object class directly, but inherits indirectly.

- In Java a class can extends to only one class at a time but it cannot extends to multiple class.

Impx - In Java each and every class must inherit Object class either directly or Indirectly

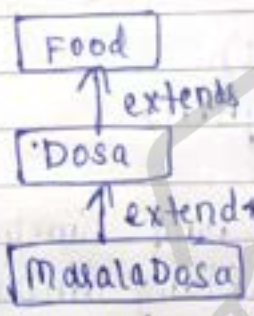
```
class Emp {   Emp by defaut extends to object class
    public static void main (String[] args){
        Emp e1 = new Emp();
        e.clone();
        e.getClass();
        e.hashCode();
    }}
```

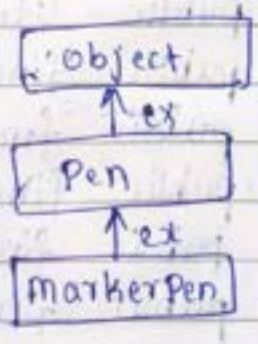
These are methods inherited by object class

(II) Multilevel Inheritance

- A class extends to another class again this class extends to another class is multilevel Inheritance

Ex:-

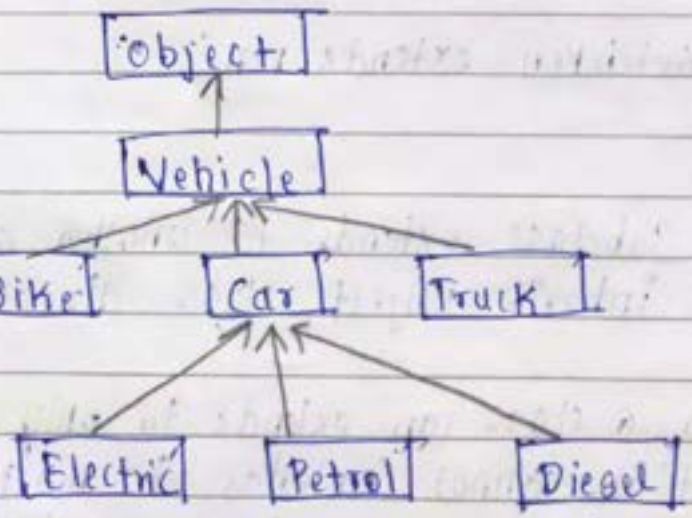




(III) Hierarchical Inheritance

- If a class contains multiple subclasses can be considered as Hierarchical Inheritance

Ex:-



Ex: Flipkart Project — main class

In Product class
States:- Name, Amount, Qty
Method:- addToCart, displayProduct, makePayment

```
class product
{
  String name;
  double price;
  ~~double~~int qty;
  void displayProduct(){
    s.o.p("product name:" + this.name);
    s.o.p("Product price:" + this.price);
    s.o.p("Product qty:" + this.qty);
  }
  void addToCart(){
    s.o.p(this.qty + " " + this.name + " product is
                    added to cart");
    s.o.p("Total amount is" + this.price x this.qty);
  }
  void makePayment(){
    s.o.p(" this.qty * this.price + " Amount paid
                    successfully");
  }
public class ElectronicProduct extends Product {
}
```

```
public class ClothProduct extends Product
{
}

public class FlipKart {
    public static void main (String [] args)
    {
        ElectronicProduct e = new ElectronicProduct();
        e.name = "Samsung S23";
        e.price = 40000;
        e.qty = 3;
        e.displayProduct();
        e.addToCart();
        e.makePayment();
        ClothProduct c = new ClothProduct();
        c.name = "Levis Shirt";
        c.price = 2000;
        c.qty = 2;
        c.displayProduct();
        c.addToCart();
        c.makePayment();
    }
}
```

05-10-2023

(iv) Multiple Inheritance:- Multiple Inherit

- A class which extends to multiple superclasses is multiple inheritance

Ex:-
```
• class A
  {
  }
• class B
  {
  }
• class C extends A, B {
  }
```

Note:- Multiple Inheritance cannot be achieved with the help of class but we can achieve by using "Interface".

Imp:- - Multiple Inheritance is not possible by using class bcz of 2 reasons.

- Ambiguity while method calling and
- Ambiguity while Constructor chaining.

Ex:-

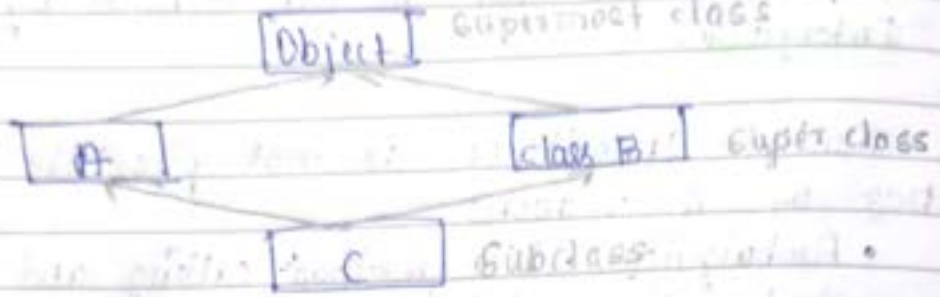
Ex:-

```
class A {                          class B {
    void run()  (OR) void run()        void run() or void jump()
    { diff logic                       }
    }                                  }
}
```

```
class C extends A, B {
    P.S.V.M() {
        C c = new C();
        c.run();
    }
}
```

- If the subclass which is extended to multiple superclass, that classes contain same methods in it then it causes Ambiguity while calling method.

☆ Diamond Problem (డైమండ్)



- In Java each and every class extends to object class either directly / indirectly. Hence all the 11 methods of object class will be inherited to all the subclasses
- Since all the methods are present in each & every class whenever the subclass try to extend more than one superclass all the 11 same object methods will be trying to inherit to subclass which causes "method Duplication" & also it causes "Ambiguity" while calling the methods

(V) Hybrid Inheritance:-
- It is a combination of Hierarchical and Multiple Inheritance.
- Even hybrid inheritance cannot be achieved by using class.

Method declaration - public void m1()

Method Body / Logic Implementation
```
{
    //logic
}
```

Method Overriding

☆ Method overriding

- It is a process of providing subclass specific implementation to the inherited methods is called method overriding.
- It cannot be achieved without Inheritance.
- We can use @override annotation to indicate the compiler about method overriding.
- It is used to achieve runtime polymorphism.

• Rules to override the method

- Method name must be same.
- Method return type must be same.
- Method signature must be same, i.e no of parameters, sequence of parameters and datatype of parameters must be same.
- Implementation change (Not a rule, optional)

Ex:-
```
class vehicle {
        void start() {
            S.O.P ("Vehicle Started");
        }
        void run() {
            S.O.P(" Press horn button");
        }
    }
    class Bike extends Vehicle {
        @Override
        void start()
        {
            S.O.P (" Bike Kick Start");
        }
    }
```

```
class car extends Vehicle {
    @Override
    void start()
    {
        S.O.P ("Push button to start");
    }
    void playMusic() {
        S.O.P ("play|pause music");
    }
}
class TestVehicle {
    public static void main (String[] args)
    {
        Bike b = new Bike();
        b.start();
        b.horn();
    }
}
```

O/P:- Bike kick Start
Press horn Button

Advantage:-
Method overriding is used to achieve "runtime polymorphism"

Note:- - Static and private method cannot be inherited hence it cannot be overridden.
- final methods can be inherited but cannot be overriden.

Ex:
```
class Bank {
    double getRI () {
```

```
    return 8.8;
  }
}
class SBI extends Bank {
    @Override
    void double getRI() {
        return 7.5;
    }
}
class HDFC extends Bank {
    @Override
    double getRI() {
        return 7;
    }
}
class TestBank {
    psvm(String[] args) {
        SBI s = new SBI();
        s.getRI();
    }
}
```

☆ Example Program for method Overriding

```
class Pizza {
    int Price;
    String name;
    Void addToppings() {
        S.o.p("No Toppings");
    void sizeOfPizza()
        S.o.p("Regular");
}
```

```
class cheezePizza extends Pizza
{
    @override
    void addToppings(){
        S.o.p(this.name + " Cheeze is loaded with cheeze");
        this.sizeofPizza();
    }
}
class PaneerPizza extends Pizza
{
    @Override
    void addToppings(){
        S.O.P(this.name + " is loaded with Paneer & Onion");

    }
    @Overide
    void sizeOfPizza(){
        S.o.P("~~Lo~~ Small"); S.O.P("Small pizza is of 8 pieces");
    }
}
class TestPizza { psvd(S[] args){
    CheezePizza c = new CheezePizza();
    c.name = "CheezePizza";
    c.addToppings();
    ~~c.sizeOfPizza();~~
    PaneerPizza p = new PaneerPizza();
    p.name = "Paneer pizza";
    p.price = 400;
    p.addToppings();
    p.sizeOfPizza();
}
}
```

o/p: CheezePizza is loaded with cheeze
Regular
PaneerPizza is loaded with Paneer & Onion
Small
Small Pizza is of 4 pieces.

06/10/2023

☆ In how many ways we can initialize instance variable
There are 5 ways to initialize the instance variable

(I) At the time of declaration

```
class Student {
        String name = "Pavithra";
}
```

Here All the created object have name "Pavithra"

(II) By using object Reference

```
class Student {
    String name; -> //declaration
    psvm(S[] args) {
        Student s = new Student();
        s.name = "Pavithra"; -> //initialization
    }
}
```

Here all objects have default null value Later it initialize individually.

(III) By using Instance Block/ Instance Initializer block
- It is also called as Object block
- It is used to initialize instance variables at the time of Object creation.
- Instance block gets executed automatically by Jvm at the time of Objects creation.
- The no of time instance block gets executed depend on the no of object created.

We can have multiple instance blocks inside a class.

Ex:-
```
class Emp{
    String name;
    {
        name = "Pavithra";
    }
    psvm()
    {
        Emp e = new Emp();
        S.O.P(e.name);   // Pavithra
    }
}
```

For multiple Instance object

```
class Emp{
    String name; int count=0;
    {
        name = "Pavithra";
    }
    {
        S.O.P("Hi");
    }
    {
        count++;
        S.O.P(count);
    }
    psvm()
    {
        Emp e1 = new Emp(); S.O.P(e1.name);
        Emp e2 = new Emp();
    }
    {
        S.O.P("Hello");
}
}
```

~~Pavithra~~ O/P

e1:
Pavithra
Hi
1
Hello

e2:
Hi
1
Hello

Drawback: Each & every object is initialized to same value and we can't change the value once it initialized.

we use this instance block when we want to assert certain configuration at the time of object creation

Ex:

```
class Mobile
{
    String color;
    {
    color = "Purple";
    }
    psvm() {
        Mobile m1 = new Mobile();
        ~~m1~~ S.O.P (m1.color); Black
        Mobile m2 = new Mobile();
        S.O.P (m2.color); Black
    }
}
```

(iv) By using Constructor — To overcome drawback of instance block — Common way of initializing

(v) By using setter() method.
↓ Standard way of initializing (API)

★ Constructor

- Constructor is a special member of a class
- Constructor name and Class name must always be same.
- Constructor doesnot have return type.

Syntax:-

```
Access_Modifier class_name (Parameter 1, ... Pa n)
{
}
```

Parameter list.

- Main purpose of constructor is to initialize instance variables at the time of object creation

Ex:-
```
class Tree{
    public Tree() {
        S.O.P("Tree");
    }
    psvm() {
        Tree t1 = new Tree();
        Tree t2 = new Tree();
    }
}
```

O/P
Tree
Tree

Ex:-
```
class Pen{ String color;
    Pen()
    {
        color="blue";
    }
    psvm(S[] arg)
    {
        Pen p1 = new Pen();
        S.O.P(p1.color);
        Pen p2 = new Pen();
        S.O.P(p2.color);
    }
}
```

☆ Types of Constructor

(I) Default Constructor
(II) Parameterized Constructor
(III) User-defined constructor

(1) Default Constructor (Imp***)

- It is automatically created by "compiler" if the user doesnot provide any constructor inside a class.

Ex: • class Mobile
{
}

$\xrightarrow[\text{generate default constructor}]{\text{Compiler}}$

class Mobile
{
Mobile() {
super();
}
}

• public class Emp
{
}

$\xrightarrow[\text{compiler}]{\text{+ve}}$

public class Emp {
public Emp() {
super();
}
}

• public class Pen
{
Pen() {
}
}

$\xrightarrow[\text{Compiler}]{}$

public class Pen {
Pen() {
super();
}
}

Not default Constructor

- The first executable Statement inside a default (or any) constructor is super calling statement.
- Access Modifier of default constructor is same as outer class. i.e if the class is public, the default constructor is public.

Imp** Note:- Default Constructor is used to initialize instance variable (Object variable) to default value based on Data Type.

(ii) Parameterized Constructor :-

- The Constructor with parameter is called Parameterized constructor.
- It is always defined by user but not by the compiler.
- The main purpose of parameterized constructor is to initialize object with different values at the time of object creation.

Ex:-
```
class mobile
{
    String color;
    Mobile(String clr) {
        color = clr;
    }
    psvm()
    {
        Mobile m1 = new Mobile("Black");
        S.O.P(m1.color);
        Mobile m2 = new Mobile("Red");
        S.O.P(m2.color);
    }
}
```

- Program to print the count of object created.

10-2023

★ Variable Shadowing / variable clashing

- In a class whenever we have local variable & instance variable with the same name then the local variable "dominates" the instance variable
- To overcome this we can use "this" keyword

to print (or) access instant variable rather than local variable

EX:-
```
class Dog
{
    String name = "Charlie";
    Double height = 8.5;
    void sleep()
    {
        double noh = 6.5;
        S.O.println("The dog sleep for" + noh +
                                "hours");
    }
    void jump()
    {
        double height = 7.5;
        S.O.println("The dog jumps" + height +
                                "height");
        S.O.println("The actual height of dog is"
                                + this.height);
    }
    public static void main (String[] args) {
        Dog d = new Dog();
        S.O.pln "The name of the dog = " + d.name);
        d.sleep();
        d.jump();
    }
```

O/p:-
```
The name of the dog is Charlie
The dog sleeps for 6.5 hours
The dog jamps 7.5 height
The actual height of dog is. 8.5
```

Ex:- class Emp

```
{
    String name;
    ~~String id;~~ int id;
    double salary;
    Emp(String name, int id, double salary)
    {
        this.name = name;
        this.id = id;
        this.salary = salary;
    }
    public static void main(String[] args)
    {
        Emp e = new Emp("Pavi", 15, 500000);
        S.o.println("Employee Details");
        S.o.println(e.name + " " + e.id + " " +
                    e.salary);
    }
}
```

If name = name; id = id; salary = salary; variable clashing

O/p:- Employee Details
Pavi 15 500000.0 null, 0 0

For variable clashing

★ Constructor Overloading

- In a class if we have more than one constructor with change in signature i.e either there should be change in no of parameters, either change in datatype of parameter or change in sequence of parameters.

Ex:- class Student

```
{
    String name;
```

```
    double perc;
    int mockrating;
    Student( string sname, double perc, int mock) {
        this.sname = sname;
        this.perc = perc;
        this.mock = mock; }
    Student( String sname, double perc) {
        this.sname = sname;
        this.perc = perc;
    }
    Student( String sname)
    {
        this.sname = sname;
    }
    psvm( String[] arg) {
        Student s1 = new Student ("Pavi", 80.2, 5);
        Student s2 = new Student ("Chandhu", 81.2);
        Student s3 = new Student ("Spandhu");
    }
}
```

Drawback:-

- In the above code there is a code redundancy where we are initializing same properties multiple time.
- We can overcome this drawback by using "Constructor chaining."

Impt:- - Constructor can be overloaded but cannot be overridden
- Constructor cannot be inherited hence it cannot be overridden.

To avoid problem of initialization. when subclass is created it might have additional attributes compared to its superclass. There could be ambuiguity/ conflicts in initializing the combined set of attributes

Imp* Why Constructor cannot be inherited.

Since the constructor and class name is same, if we inherit the constructor will be violating the rules of the constructor.

Hence constructor cannot be inherited.

★ Copy Constructor

— It is used to create a new object by copying the contents of existing object. It takes object of the same class as parameter.

```
Ex:- class Emp {
        String ename;
        ~~double~~ String designation;
        long salary;
        Emp(String ename, String designation, long salary)
        {
            this.ename = ename;
            this.designation = designation;
            this.salary = salary;
        }
        Emp(Emp other_Emp) {
            this.ename = other_Emp.ename;
            this.designation = other_Emp.designation;
            this.salary = other_Emp.salary;
        }
    class TestEmp
    {
        psvm()
        {
            Emp e1 = new Emp("pavi", "Developer", 50000
            S.o.p(e1.ename+" "+e1.designation+" "+
```

http://bit.ly/exams

```
e1.salary);
Emp e2 = new Emp(e1);
S.o.pln(e2.ename + " " + e2.designation + " " +
        e2.salary);
}
}
```

O/p:

09-10-2023

☆ Super Keyword

Super Keyword is used to access immediate super class, states and behaviour

Ex:- To access Super Class States (Instance variable

```
class VTU
{
    int fee = 2500;
}
class College extends VTU {
    int fee = 75000;
    college (int fee) {
        S.o.pln("Placement fees:" + fee);   - Local
        S.o.pln("College fees:" + this.fee);   - Instance
        S.o.pln("VTU fees:" + super.fee);   - Super class variable
    }
}
class Test {
    psvm() {
        College c = new College(2500);
    }
}
```

Access

o/p:- Placement fees: 2500
College fees: 75000
VTU fees: 25000

Note:- Super keyword is used to access super class instance variable & non-static methods but not static methods and static variable.
- super keyword cannot be used in static block & static methods.

Ex:- To access non-static method

```
class Father {
    void drink() {
        S.O.P ("Drinks Alcohol");
    }
}
class Son extends Father {
    @override
    void drink() {
        S.O.P ("Drinks Horlicks");
    }
    void goingToTrip () {
        S.O.P ("Going to trip with friends");
        super.drink();
    }
}
class Test {
    Son s=new Son();
    s.drink();
    s. goingToTrip();
}
}
```

O/p :- Drinks ~~Alcoholicks~~
Going to trip with friends
Drinks Alcohols.

☆ Write a difference between this and super keyword

this keyword	super keyword
- It is used to access the state & methods of current invoking object - It represent the current invoking object of the class	- It is used to access the super class states & variables - It represent the current invoking object of the parent

☆ As-A Relationship
└ Association
- Association is a process of one object getting associated with another object.
- It is one of the commonly used object oriented feature.
- Association is also known as As-A Relationship.
- There are two types of association
(I) Composition
(II) Aggregation

(I) Composition
- It is a special form of association where an associated object cannot logically or independtly exist without an owner object, i.e if the owner object

get destroyed then the associated object also get destroyed.

- Composition is also known as Strong As-A relationship.

Ex: Real world ex - Fish and water, Pop-up window and Button, Car as a Engine.

```
Ex:- class Engine()
    {
        String name;
        Engine (String name) {
            this.name = name;
        }
        void StartEngine() {
            S.o.p(name + " Engine is started");
        }
    }
    Class Car {
        String name;
        Engine e;
        Car (String name)
        {
            e = new Engine("Turbo 1.2"); // Engine object
            this.name = name;
        }
        void startCar()
        {
            S.o.p(name + " Car get started");
            e.startEngine;
        }
    psvm() {
        Car c = new Car("Hyndai");
        c.startCar();
    }
}
```

(ii) Aggregation

- It is a special form of association where an associated object can logically / independtly exist without an owner object.
- Aggregation is also known as weak IS-A relation -ship.

Ex:- Company and Employee, Student and Course,

Ex:- For student & Course

```
class Course {
    String cname;
    Course (String cname) {
    this.cname = cname;
    }
}
class Student {
    String sname;
    Course c;
    Student (String sname) {
    this.sname = sname;
    }
    void enroll(Course c) {
        S.o.p (sname + "Enrolled for: " + c.cname+
                " course");
    }
}
class Test {
    Psvm ()
    {
        Course c1 = new Course ("Java");
        Course c2 = new Course ("Web");
        Student s1 = new Student ("Pavi");
```

```
Student s2 = new Student ("chandhu");
s1.enroll(c1);
s2.enroll(c2);
}
}
```

10/10/2023

(IMP***)

★ Encapsulation

- It is a process of wrapping up of data and code into a single unit with the help of data handling methods such as setter() & getter().
- Java bean classes are known as fully encapsulated class.
-

• Java bean class Specifications [Encapsulated class specif]

- The class must be public non-abstract class.
- The class should contain "private data members".
- The class should have setter() and getter() method
- The class should contain zero parameter constructor or default constructor or parameterized.
  ↳ To call superclass constructor.

• Advantage of Encapsulation

- We can achieve "data hiding".
- We can perform data validation.
- We can make data read only or write only.
-

Ex:-

```
public class Student {
    private String name;
    private double perc;
    private int age;
```

```
public void setName(String name) [For each Private
{                                 member there should be
    this.name = name;             public setter & getter()]
}
public String getName() {
    return this.name;
}
public void setPerc (double perc) {
    if (perc ≥ 35 && perc <= 100)   //data validate
    {
        this.perc = perc;
    }
    else
    {
        S.o.pln("Your age are not eligible");
    }
public double getPerc () {
    return this.perc;
}
public void setAge(int age) {
    if (age ≥ 18 && age < 50) {
        this.age = age;
    }
    else
        S.o.p("your age is not eligible");
}
public int getAge() {
    return this.age;
}
}
public class Test {
    psvm(S[] args) {
```

```
Student s = new Student();
s.setName("Pavi");
s.setAge("22");
s.setPerc("80");
S.O.println(s.getName() + " " + s.getAge() + " "
        + s.getPerc());
    }
}
```

Project:-

1) Create an 'Employee' class with the following private fields
- 'employee id' (int)
- 'name' (String)
- 'age' (int)
- 'salary' (double)

2) Implement a Constructor in the 'Employee' class to initialize these fields when a new employee object is created.

3) Provide public getter and setter method for the 'age' and 'salary' fields to ensure controlled access to these properties. The 'employeid' and 'name' should not have setter.

4) Implement a 'displayInfo' method in the 'Employee' class that displays the employee name, employeeId age and salary.

5) In the Main class, create several instance of 'Employee' class, set their attributes using setters and display

their information using the 'displayInfo' method.

6] Ensure that the employee's 'employeeid' is automatically generated and assigned when a new object is created and it should not be modifiable from outside the class.

7] Ensure that 'age' and 'salary' values are within valid range (Ex. age should be b/w 0 & 120, salary should be non-negative)

8] Demonstrate encapsulation by accessing & modifying the employee's attributes through the provided getter & setter methods, not directly accessing the private fields.

Bonus (Optional)

a] Implement validation in the setter methods to ensure that the 'age'

```
public class Employee {
    private int eid;
    private String name;
    private int age;
    private double sal; private static int count = 1005;
    Employee(String name, int age, double sal)
    {
        this.name = name;
        this.age = age;
        this.sal = sal;
        this.eid = eid;
    }
    public void setAge(int age) {
        if (age > 0 && age < 120) {
            this.age = age;
        }
    }
```

```
public int getAge() {
    return this.age;
}
public void setSal(double sal) {
    if (sal > 0) {
        this.sal = sal;
    }
public void double getSal() {
    return this.sal;
}

public void displayInfo()
{
    S.O.println("Emp name: " + this.name + "Emp age: " +
    this.age + "Emp sal: " + this.sal + "Emp id: " +
                this.id);
}
}

Public class Test {
    psvm() {
        Employee e1 = new Employee("Pavi", "21", "60000")
        S.O.P("Employee details");
        e1.displayInfo();
        S.O.P("Updated details");
        e1.setAge(22);
        e1.setSal(70000);
        e1.displayInfo();
    }
}
```

Note:- Private data members with only setter method without

getter method is called write only data.

Ex:-
```
public class Emp{
    private String name;
    public void setName(String name){
        this.name = name;
    }
}
public class Test {
    psvm(){
        Emp e1 = new Emp("XXX");
        e1.setName("XXX");
        S.O.P(e1.name); (It is not visible
    }
}
```

- The private data members with only getter method without setter method is read-only data.

Ex:-
```
public class Emp{
    private String name = "Pavi";
    public String getName( String name){
        return name;
    }
}
```

---

* Constructor chaining (Imp**)

- Calling one constructor inside another constructor by using this() and super() calling statements is called constructor chaining.
- A Constructor can be called only inside another constructor by using this & super calling statement but not using constructor name.

- Calling Const another Constructor must be first executable statement inside any constructor.
- this () calling statement is used to called current class overloaded constructor.
- A Constructor can call only one constructor inside another constructor.
- Inside a constructor either super or this must be first executable statement but not both.

Impx - It is mandatory to call immediate superclass (either 0 parameter / parameterized) constructor inside any of the subclass constructor.

Impx - If the user doesnot call any constructor inside a another constructor then by default compiler will call immediate superclass '0' parameter constructor [no argument constructor]

Ex:- (1)
```
student (int a)
{
    this.a=a;
}
Student (String s, int a) {
    this.s = s;
    this (a); // Error
}
```

(11)
```
Student (int)
{
    this (100);
    super (a);
}
student (int a)
{
    this.a=a;
}
```

- ~~Constructor callin~~
- Cyclic constructor calling is not possible in java.

Ex:
```
Emp()
{
    this(100);
}
Emp(int a)
{
    this();
}
```

Example for constructor chaining

```
class Student {
    String name;
    double perc;
    int mockRating;
    Student( String name, double perc, int mockrating)
    {
        this (name, perc);
        this.mockRating = mockRating;
    }
    Student (String name, double perc)
    {
        this (name);
        this. perc = perc;
    }
    Student (String name)
    {
        this.name = name;
    }
    psvm() {
```

```
    Student s1 = new Student ("dinga", 45.25);
    Student s2 = new Student ("guldu");
    Student s3 = new Student ("munna", 6225, 8);
  }
}
```

- Constructor chaining by using super() statement
  super() is used to call immediate super class constructor

Ex 1:-
```
class Human {
    String name;
    int age;
    Human (String name, int age)
    {
        this.name = name;
        this.age = age;
    }
}
class Employee extends Human {
    int empno;
    double salary;
    Employee (String name, int age, int empno, double
                                salary)
    {
        super(name, age);
        this.empno = empno;
        this.salary = salary;
    }
}
class Student extends Human {
```

```
    int rollno;
    double perc;
    Student (String name, int rollno, double perc,
                int age) {
    super(name, age);
    this.rollno = rollno;
    this.perc = perc;
  }
}
class Test
{
    psvm (S[] a) {
        Employee e = new Employee("Pavi", 22, 300,
                                    7000000);
    }
}
```

Ex2:

```
class Tree
{
    double height;
    Tree (double height) {
        this.height = height;
        S.o.p("I am double param");
    }
    Tree()
    {
        super();
        S.o.pln("Beta");
    }
}
class cocTree extends Tree {
```

```
coctree (double d)
{
    this ("super()");
    S.O.p ("Alpha");
}
coctree (String s)
{
    super();  By default it call Tree ()
    S.O.p ("Gama");
}
coctree () ——— (1)
{
    this (45.25);
}
}
class Test
{
    psvm( S[] args)
    {
        CocTree c = new coctree();
    }
}
```

O/p:- Beta
Gama
Alpha.

Ex-3:-
```
class Vehicle {
    String registerno;
    Vehicle ( String rno) {
        this. registerno = rno;
    }
}
```

```
class Bike extends Vehicle
{
    void start() {
        S.o.p("Bike Kick start");
    }
}
```

o/p: Compilation error

8-10-2023

☆ Variable assignment (Type casting)

Converting value of one datatype into another datatype.

(i) Primitive Casting

(ii) Non-primitive Casting

(i) Primitive Casting

* Implicit primitive casting [Widening]
* Explicit primitive casting [Narrowing]

* Implicit Casting L ≥ R

- It is a process of converting lower datatype into higher data type.
- Implicit Casting takes place automatically.

Ex:-
```
int i = 100;
double d = i; //implicit

short s1 = 250;
long l1 = s1; //implicit
```

* Explicit Casting

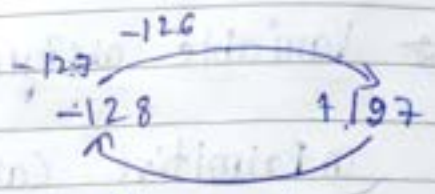
- It is a process of converting higher datatype into lower datatype.
- LHS < RHS

- Explicit casting has to be performed manually.
- whenever the LHS is lesser than RHS then we have to perform explicit casting

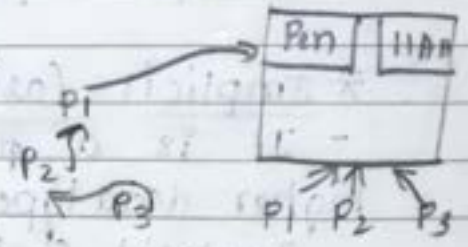
Ex:- • int a = 350;
short b = short(a);
short s = a // error

• long l = 500; → int a = l; // error
int a = int(l);

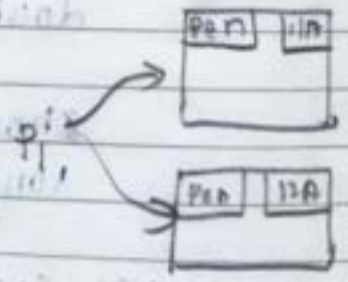
• int a = 130;
byte b = byte(a);
S.O.P(b);
O/P = -126



Note:- In java an object can have multiple reference
Pen p1 = new Pen();
Pen p2 = p1;
Pen p3 = p2;



- In java we cannot have multiple object with same reference
Pen p1 = new Pen();
Pen p1 = new Pen();



Garbage Collection:
- In java garbage refers to unused object (Dereferenced object)

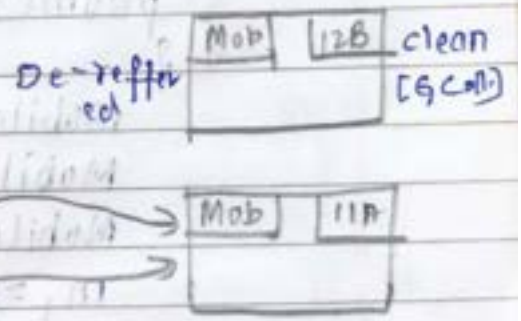
- Garbage collection is a process of reclining runtime unused memory.
- In java we cannot delete the java explicitly but we can make a object eligible for garbage collection by de-referring the objects.

Advantage :-
- Memory becomes more effecient bcz the garbage collector removes the de-referred object (unused object) from the heap memory
- It is automatically done by the JVM, so that we don't need to de-allocate any memory.

- There are 3 ways to ~~defer~~ de-refer an object.

1) By assigning any new object to the existing reference.

Ex:- Mobile m1 = new mobile();
Mobile m2 = new Mobile();
m1 = m2;



2) By assigning null to the reference
Mobile m1 = new Mobile();
m1 = ();

3) By creating Anonymous Object.
new Mobile();

finalize():
- This method is used to indicate the object is de-reffered. Hence the object is eligible for Garbage Collection.

- This method is used in clean up process
- System.gc():- It is used to invoke garbage collector.

gc() method is present in System class & runtime class which is present in ST.java.

- finalize() method is present in Object class.

Ex:-
```
public class Mobile {
    @Override
    protected void finalize() throws Throwable
    {
        S.o.p(this + "Object is ~~eligig~~ eligible for
                Garbage collection);
    }
    psvm()
    {
        Mobile m1 = new Mobile();
        Mobile m2 = new Mobile();
        Mobile m3 = new Mobile();
        m1 = null;
        m2 = null;
        m3 = null;
        System.gc();
    }
}
```

o/p

```
Mobile@7c9d1590 object is eligible for GC
Mobile@79d1c680 object is eligible for GC
Mobile@257af473 object is eligible for GC
```

(II) Non-primitive Casting:- (Complex)

There are two types

* Up-Casting
* Down-Casting

] For these, it requires inheritance

* **Up-Casting:-**

The process of providing Superclass reference to the subclass object, i.e reference from super class, object from subclass.

Ex:- Fruit f = new Apple();
Food fo = new Idly();
Player p = new CricketPlayer();
Engineer e = new CivilEngineer();
Animal a = new Cat();
Vegetable v = new Carrot();
Vehicle Vh = new Car();

Pen p1 = new Pen();
object ob = p1; //Upcasting
Pen p2 = new Pen();
Object ob1 = p2; //Upcasting
Sketchpen sp = ob1; //error
Subclass Superclass

Object
↑
Pen
↑
Sketchpen

Upcasting is implicit, we dont need to cast explicitly

* **Down-Casting:-**

For the above ex: SketchPen Sp = SketchPen(ob);
Syntatically & logically correct

Vehicle
Bike Car

Vehicle v1 = new Car(); //Upcasting
Object ob = v1; //Upcasting
Vehicle v2 = new Bike(); //Upcasting
Bike b1 = v2; // Error
Bike b1 = (Bike) b2; // Downcasting

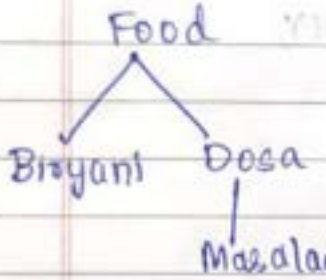
```
Bike b1 = (Bike) v2;
Car c1 = (Car) v1;
```
] syntatically & logically correct

```
Bike b1 = (Bike) v1;
Car c1 = (Car) v2;
```
] syntatically correct but logically wrong

↓

During runtime classCastException appeared

When even the downcasting is syntatically correct but logically wrong then we get a exception called **ClassCastException (CCE)**.

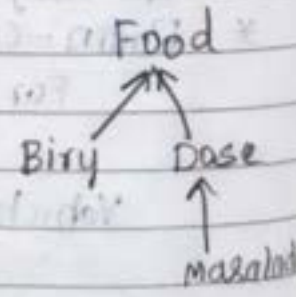



```
Dosa d = new Masaladosa();
Food f = d;
Object ob = f;
Masaladosa md = d; //error
Masaladosa md = (Masaladosa) d; //Down Cast
Biryani b = (Biryani) f; //CCE
```

★ **Instanceof:-**

- It is used to check the **object type**.
- Instanceof Keyword always evaluates for true or false condition.

Ex:-
```
class Food {
}
class Dosa extends Food {
}
class Masaladosa extends Dosa {
}
class Biryani extends Food {
}
```



```
Class Test {
    psvm(){
      Dosa d = new Masaladosa();
      Food f = new Biryani();
      Food f1 = d;
      Object o = f;
    if (f instanceof Masaladosa)
    {
      S.O.p("Down-casted to masaladosa");
      Masaladosa md = (Masaladosa) f;
    }
    else if (f instanceof Biryani)
    {
      Biryani b = (Biryani)f;
    }  S.O.p("Down-casted to Biryani");
    else
      S.O.p("Invalid");
  }
}
```

o/p:- Down-casted to Biryani

Imp ☆ Characteristics of Up-casting:-

- In case of Up-casting using superclass reference we can access inherited variables & methods, but not subclass specific variables and methods.

Ex:-
```
class vehicle {
        int modeno;
        String name;
        void start() {
        }
}
```

```
class car extends Vehicle {
        int nog;
        void playmusic() {
            S.O.ptln("Play/pause music");
        }
}
class test {
    psvm() {
        Vehicle v = new Car();
        v.modelno = 10;
        v.name = "xxx";
        v.start();
        v.playmusic(); //error
        v.nog = 5; //error
```

For this Car c = (Car) v; Downcasting

```
        c.nog = 5;
        c.playmusic();
```

- In case of upcasting when we invoke overridden method by using super class reference then the implementation will be executed from sub class but not from the superclass.

Ex:-

```
Class Parent
{
    void m()
    {
        S.O.p("Parent");
    }
}
class child extends Parent {
    @Override
    void m() {
```

```
    S.O.P("child");
  }}
  class Test {
      psvm(){
          Parent p = new Child();
          p.m();
      }
  }
```

O/P:- Child.

Ex:-
```
class Player{
    void m(){
        S.O.P("Player play a game");
    }
}
class CricketP extends Player{
    @Override
    void m(){
    S.O.P("Cricket player plays cricket");
    }}
class FootballP extends Player{
    @Override
    void m(){
    S.O.P("Football player plays Football");
    }
}
psvm(){
    Player p = new Player();
    Player p1 = new CricketP();
    Player p2 = new FootballP();
    p.m();
    p1.m();
    p2.m(); }}
```

O/P:

o/p:- Player play a game
Cricket player plays cricket
Football player plays football.

16/10/2023

★ Method

There are two types of method in java
(i) Concrete method
(ii) Abstract method

(i) Concrete method
- A method with declaration and implementation (logic) is called concrete method.

Ex:-
```
void m() ——— declaration
{
    // logic  Implementation
}
```

- A concrete method can be declared inside a class & abstract class.

(ii) Abstract method
- A method with only declaration without any implementation is called abstract method.
- Abstract method is always declared with abstract keyword.
- Abstract method must be terminated with semicolon.
- Abstract method can be declared only inside abstract class and interface. but not inside normal class (concrete class.

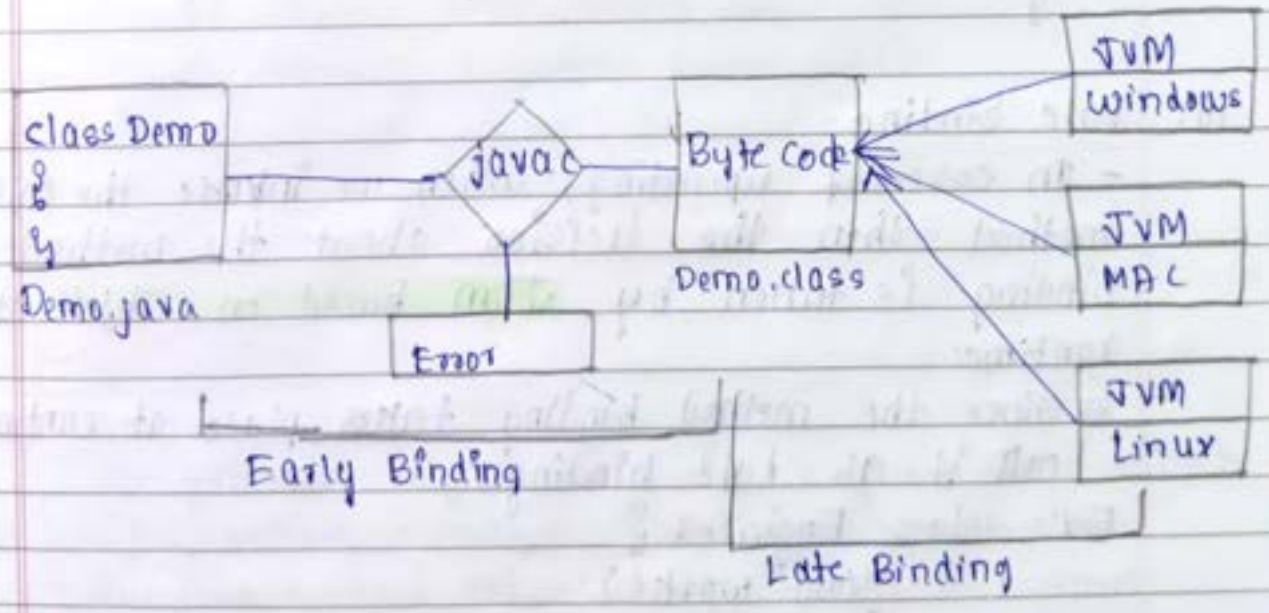
Ex:-
```
abstract void m(); declaration
    // no logic
```

☆ Method binding

- It is a process of attaching method logic/implementation to its method call.

Ex:-
```
class Mouse {
    void rightClick() {
        //logic
    }
    psvm()
    {
        Mouse m1 = new Mouse();
        m1.rightClick();
    }
}
```

- There are two types of method binding
(I) Early binding / static binding
(II) Late binding / dynamic binding



(I) Early binding
- In case method overloading, when we invoke the

overloaded method then the decision about the method binding is taken by the "compiler" based on parameter and argument

- Since the method binding takes place at compile time we called as Early binding.

Ex:-
```
class website {
        void login (String uname, String pwd)
        {
        }
        void login (long contact, int otp)
        {
        }
}
class TestWeb {
    psvm() {
        Website w = new Website();
        w.login(9353723547l, 1234);
    }
}
```

(ii) Late binding

- In case of upcasting, when we invoke the overridden method then the decision about the method binding is taken by JVM based on object at the runtime.

- Since the method binding takes place at runtime we call it as late binding.

Ex:-
```
class Engineer {
        void work()
        {
            S.o.p("Engg work");
        }
}
```

```
class SwEng extends Engineer {
    @override
    void work() {
        S.o.P("Test/develop/SW");
    }
}
class civilEng extends Enggineer {
@Override
void work() {
    S.o.p("Construct bridge & dam");
}
}
class Test {
psvm() {
    Engineer E = new Engineer();
    Engineer se = new SwEng();  -> Upcasting.
    Engineer cv = new CivilEng();
    e.work(); // Eng work
    se.work(); // Test/develop/SW
    cv.work(); // Construct bridge & dam
}
}
```

★ Polymorphism

- In general Polymorphism is a greek word which means Poly - many, Morphism - Forms.
- In java method takes multiple forms is called polymorphism.
- Polymorphism is a same task (or) action performed in different ways.

- Types of Polymorphism
(I) Compile time / Static polymorphism
(II) Runtime / dynamic polymorphism

(I) Compiletime Polymorphism
- In java, Compiletime polymorphism can be achieved by using method overloading.
- Method overloading is nothing but same activity performed by taking inputs.
- Compiletime Polymorphism is also known as Early binding.

(II) Runtime Polymorphism / Virtual / Dynamic Polymorphism.
- It is a ability of a method to behave differently based on the object. is runtime polymorphism.
- Runtime polymorphism can be achieved by using method overriding.
- Runtime polymorphism is also known as late binding.

Note:- Without inheritance it is not possible to achieve runtime polymorphism.

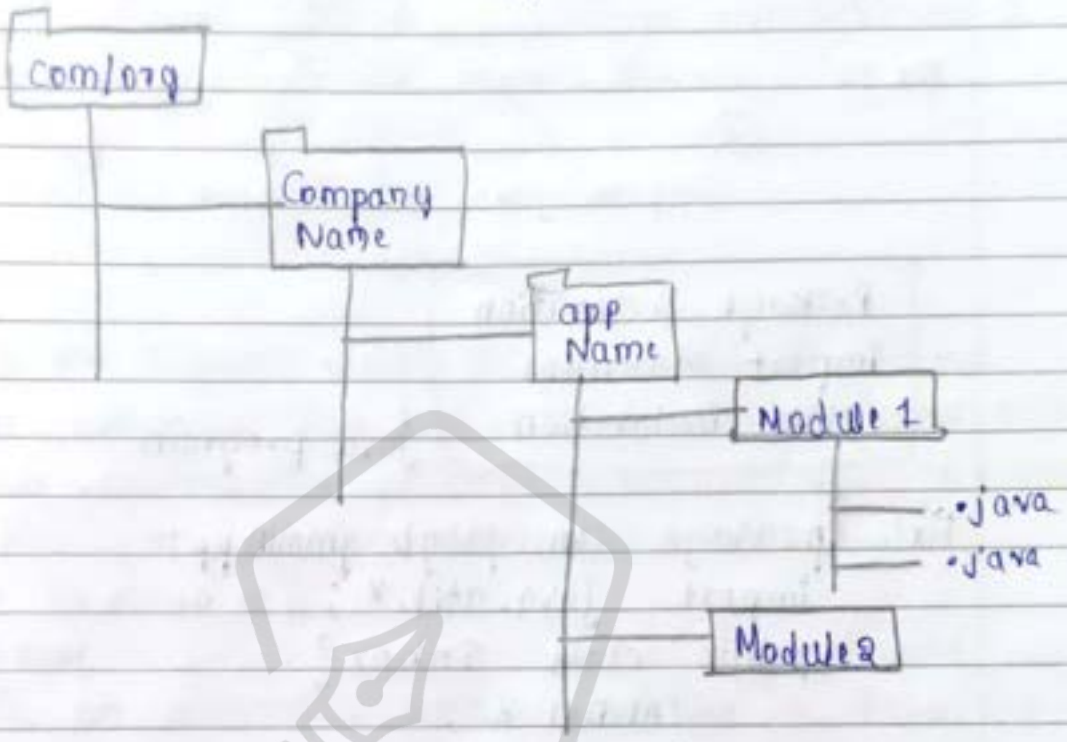
17/10/2023

★ Package
- Package is a simple folder or directory which is a collection of similar java resource.
- Package are used to organise and group the related classes & interfaces into an hierarchical structure

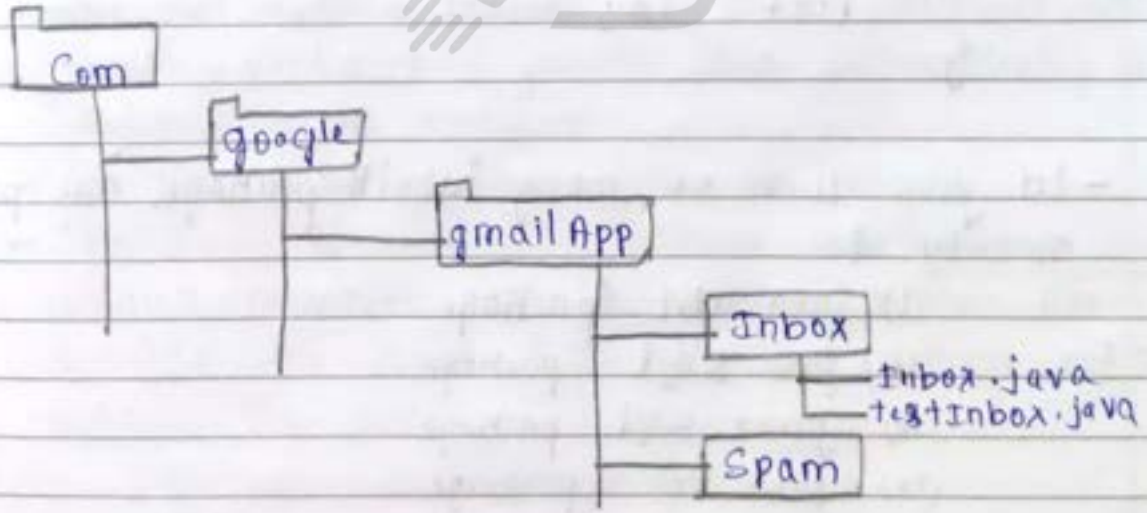
Advantage
- It is easy to achieve modularity

- Better maintenance of project.
- Easy to access and search.

Standard Structure of Package



Ex:-



- Public class and members can be accessed within the class, outside the class, within the package and also outside the package.
- when we are accessing public members outside the package then it is mandatory to use fully qualified

class name or import statement.

Class name for Inbox :- Inbox
Fully classified class name :- com.google.gmail.Inbox

For ex :- java.util.scanner sc = new java.util.scanner(System.in);
(or)
import java.util.Scanner

Package Declaration
Import Statement
class Declaration

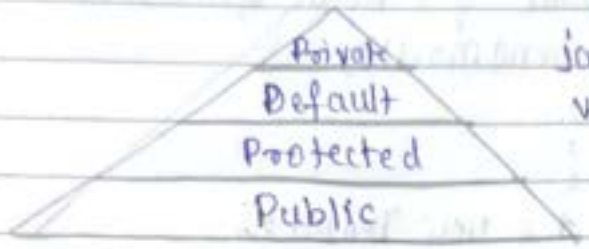
java program

Ex:
```
package com.google.gmailApp;
import java.util.*; // to access all classes present in
public class Inbox{                 java.util package
    Scanner sc;
    List lb;
}
```

- In java there are many inbuilt packages are present namely :
  (i) java.util package
  (ii) java.sql package
  (iii) javax.sql package
  (iv) java.io package
  (v) java.lang package
  └ String, Object, System
- java.lang package is a <u>default</u> package (It is available for all other package without importing).
- The classes and interfaces present in java.lang package will be available for all the packages without any

import statements

☆ Access Modifier



The access type of java members or a vis

(i) Public :-

- It is the highest visible access modifier in Java.
- Public members can be accessed within class, outside the class, within the package & also outside the package.
- All the java members can be public.

(ii) Default :-

- It is an automatic access modifier in Java, i.e if the user doesnot provide any access modifier like public, protected & private then automatically it takes default access modifier.
- All the java members can be default.
- The scope ~~of~~ default members are "package level", i.e default members can be accessed within class, outside the class & within the package but not outside the package.

Ex:-
```
package com.google.GmailApp;
public class Gmail {
    public void SendMail() {
        S.o.pln("Mail sent through gmail");
    }
}

package com.microsoft.Team;
```

```
import com.google.gmailApi.Gmail;
public class Team {
    void Schedule_Meet() {
        Gmail g = new Gmail();
        g.sendMail();
    }
    psvm() {
        Team t = new Team();
        t.Schedulemeet();
    }
}
```

Note:- If the sendMail() is default, then it is not possible to access outside the package either by import statement / fully classified classname.

(III) <u>Private</u>

- It is the least visible access modifier in Java.
- The Scope of private members is class level, i.e Private members can be accessed within the class but not outside the class.
- Private methods members cannot be inherited.
- Outer class cannot be <u>private</u>.

Ex:-
```
public class Shoes {
    private void flyingShoes()
    {
        S.o.p("Logic for flying shoes");
    }
}
public class Test {
    psvm() {
        Shoes s = new Shoes();
```

```
    s.flyingShoes();  // The method flyingShoes() from Shoes
    }                 is not visible bcz it is declared as
}                     private.
```

Error

```
Ex:- Class Outer_Class{ //It can be public, default

        Class Nested_class{ } //Public, default, private, protected
        Void m()
        {
            class Inner_class{ //No Access Modifier
            }
        }
    }
```

(iv) Protected

- Protected members can be accessed within the class, outside the class, within the package and outside the package.
- To access protected members outside the package we have to perform inheritance., without inheritance it is not possible to access the protected members outside the package.
- Outerclass cannot be protected.

Note:- Protected members can be accessed outside the package using inheritance by creating subclass object but not by creating superclass object.

```
Ex:- Package Package1;
     public class Parent {
```

```
    protected void m(){
        S.o.p("Parent class protected m()");
    }
}
```

```
package Package2;
import Package1.Parent
class child extends Parent{
    psvm(){
        Child c = new Child();
        c.m();
        Parent p = new Parent();
        p.m(); // Error
    }
}
```

Modifiers	outerclass	Nestedclass	method	variable	Constructor
public	✓	✓	✓	✓	✓
private	✗	✓	✓	✓	✓
default	✓	✓	✓	✓	✓
protected	✗	✓	✓	✓	✓

Note: - In case of method overriding in subclass the method name must be same, return type must be same, signature must be same but access modifier can be same (or) it can higher visible access modifier.

protected	default	public	private
↓	↓	↓	↓
protected public	default protected public	public	It can't be inherited So, No overriding

Inheritance V/S Accessing

★ final

- It is a keyword which is used to declare java members like class, variable, method.

final
- class
  - Can't be extended (NO subclass)
  - Ex:- System, Scanner, String
- method
  - Can be inherited but we can't override
- variable
  - can be inherited but can't change the value [Constant]

- There are many inbuilt final classes are present in Java Namely Scanner class, String class, System class, Integer, StringBuffer, StringBuilder, Double

★ How to avoid Inheritance for a class?

- By declaring the class as a final.
- By making the constructor of a class as a private

19-10-2023

★ static

- It is a keyword which is used to declare a variable, method, block and nested class.
- Static members are also known as class members.
- In java static members gets a memory at the time of "class loading".

- Class loading:-

It is a process of bringing class file into jvm memory.

- Static members get created in class area (or) method area. (meta space)
- Static members will be loaded only once.

- We can avoid object creation to access static members
- If the variable is declared with static keyword then if any modification is made for that variable then we can't get back the initial values, i.e we can get the recent modified value.

Why main method is static?

- main method will be loaded once.
- Main method will be loaded at the time of class loading.
- JVM can avoid object creation to access the main method.

☆ static() :-

- when we declare a method with static keyword then the method cannot be inherited but it can be accessed with the subclass reference.

Ex:-
```
public class Parent {
    static void m() {
        S.o.p("Parent class Method");
    }
}
public class child extends Parent {
    @Override
    static void m() { //error static method cannot
        S.o.p("child method"); be overriden
    }
    or
    static void m() { //No error, b/c no override
        S.o.p("child method"); notation
    }
}
```

Note:- Static methods cannot be inherited so, it can't be overridden.

Ex:-
```
Public class Parent{
    static void m(){
        S.o.p("Parent method");
    }
}

Public class child extends Parent{
}
public class Test
{
    public static void main(S[] args)
    {
        Parent.m();
        (or)
        child.m();
    }
}
```

* Method hiding
  - Declaring superclass methods as a static is called method hiding.
  - We can access Super Class static method by using Superclass reference (or) by using subclass reference.
  - Static method can be overloaded.

Imp**
* Static block | class block
  - A block declared with static keyword is called static block

```
static
{
    ----
}
```

- static block will be executed at the time of "class loading before the main method"

Ex:-
```
class StaticDemo{
    static
    {
        S.O.p("Static Block"); ——— ①
    }
    psvm(S[] args)
    {
        S.O.P("main method"); ——— ②
    }
}
```

O/p:- Static Block
main method

Note:- Static block will be used to write a configuration code (or) set of code which has to be executed at the time of class loading (Application loading).

For ex:- ~~At the~~ when we install the Facebook Applic at the time of application loading, connectivity with the database will be done, such code / configuration will be present inside static block

- static blocks will be executed by "class loader" at the time of class loading (when the class is loaded into memory)
- static block is executed only one once for a given class, regardless of how many instances of that class is created.

20-10-2023

★ When we need to go for static method?
- Whenever a method doesnot exhibit object specific behaviour or polymorphic behaviour then we can go for static methods.

★ toString()
- toString() is a method which is present in super most class called "Object class".
- To represent the object in the form of String we use toString().
- toString() is used to print the content of the object instead of Object Address.

Ex:-
```
public class Emp {
    @Override
    public String toString()
    {
        ~~S.o.p("Pavi");~~ return "Guldu";
    }
    public static void main(String[] args){
        Emp e1 = new Emp();
        S.o.p(e1); //Here without toString() it points the object address. But toString() m
        Emp e2 = new Emp();
        S.o.p (e2); // Pavi
    }
}
```

O/p:- Pavi
Pavi

Syntax:-
```
public String toString()
{
    return String_value;
}
```

Local variable can't have access modifier

System.out
↓
class static field

out, err, in
static field

Ex:- public class Student {
String sname;
int rno;
double perc;
public Student(String sname, int rno, double perc) {
this.sname = sname;
this.rno = rno;
this.perc = perc;
}
@Override
public String toString() {
return this.sname + " " + this.rno + " " + this.perc;
}
public static void main (S[] args) {
Student s1 = new Student ("Pavi", 098, 75.2);
Student s2 = new Student ("Chandu", 022, 77.1);
S.o.p(s1); S.o.p(s2);
} }

o/p:- Pavi 098 75.2
Chandu 022 77.1

Note:- By default println statement internally calls toString() when we pass object as a argument
- If we don't override the toString() in the any class then if pass object as an argument in the print statement then it will execute the toString() method of object class

15-10-2023

Abstract class

- A class which is declared with abstract Keyword is called Abstract class.
- Abstract class is like a normal class which can have variable, methods, constructor but the only change is it can have abstract method & we can't create object for abstract class.
- Abstract class is also known as non-primitive datatype which is used to refer subclass object. (we can't refer same class object [Abstract class]).
- The abstract() which is present in abstract class will be overridden in subclass.
- Abstract class and Interface is used to achieve "abstraction"
- By using abstract class we can't able to achieve 100% abstraction.
-

Note:- A class can be declared as a abstract without any abstract methods, but if the class contains abstract method then it is mandatory to declare a class as a abstract

- Can we oberload an abstract()?
  Yes, we can overload abstract()

Ex:- public abstract void add(int a);
public abstract double add(int a, double b);

- Can we override an abstract()?
  Yes, we have to override the abstract().

- What we have to do if we are not ready to override the abstract() in subclass?
  We need to declare subclass as a abstract

- Is it possible to have abstract class without any abstract method
  Yes
  Ex:- Http Servlet (Abstract class with only concrete methods)

Ex:-
```
abstract class Keyboard {
    void PressQ()
    {
        S.o.p.ln("Press Q");
    }
    void PressP()
    {
        S.o.p ("Press P");
    }
    abstract void PressEnter();
}
class NotePad extends Keyboard {
    @Override
    void PressEnter()
    {
        S.o.p("Goes to Nextline");
    }
}
class MusicPlayer extends Keyboard {
    @Override
    void PressEnter()
    {
```

```
        S.O.P ("Play/Pause the music");
    }
}
Class TestClass{
    psvm(){
        Keyboard K = new Keyboard (); //Error.
        Keyboard K1 = new NotePad();
        Keyboard K2 = new MusicPlayer();
        K1.PressEnter();
        K2.PressEnter();
    }
}
```

O/p: Goes to NextLine
Play/Pause the music.

-10-2023

* **Interface** (Imp***)
  - It is an intermediate b/w consumer and the service provider
  - It is always declared with "interface" Keyword
  - It contains abstract methods which is used to achieve abstraction
  - Since interface contain only abstract method we can achieve 100% abstraction
  - The subclass can inherit the interface by using "implements" Keyword.

```
Ex:- public interface Inf1{
        abstract void m();
    }
```

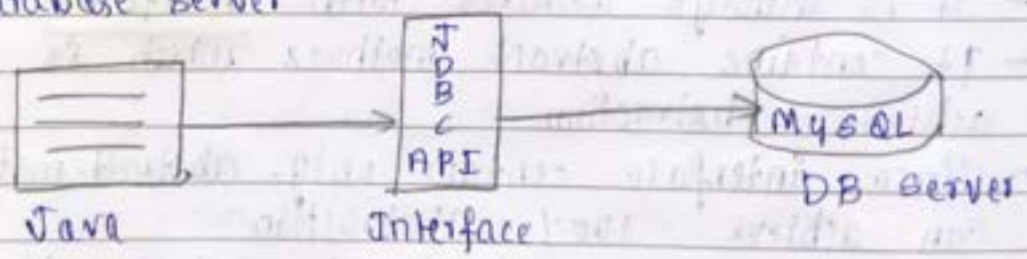
```
public class Demo implements Inf1 {
    @Override
    void m() {
    }
}
```

- The class which inherit interface is called "implementation class".
- An abstract class and interface also acts as non-primitive data type
- we can't create object for interface, but interface can be used refer implementation class objects
- Interface doesnot have any constructor

Real world Example:- Switch is interface b/w User and light, Remote is an interface b/w User and Tv/Ac/Projector. Regulator is interface b/w User & Fan

Application Specific Example for Interface

JDBC API is an Interface b/w Java program & Database server

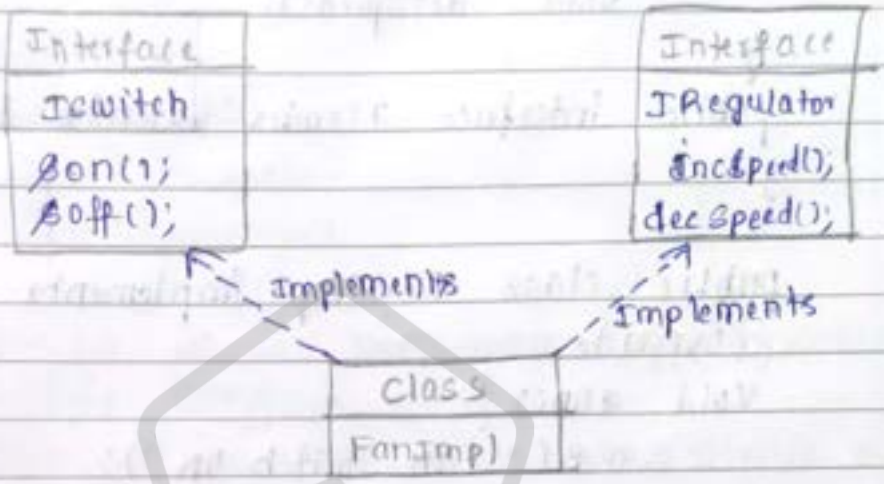


- Interface is an coding contract for subclasses.
- Abstract class can also inherit the Inherit
- In Interface if we terminate any method by default it takes public abstract method.

Ex: public interface ISwitch {

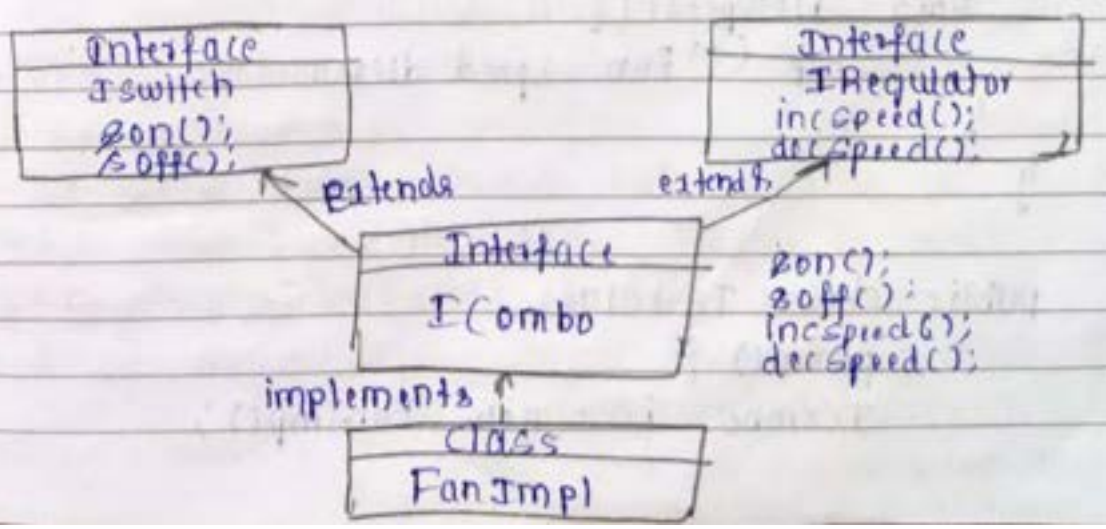
```
void SON();  → By default public abstract void SON();
void SOff();
}
```

- We can achieve multiple inheritance with the help of interface.



- To inherit
  - class to class :- extends
  - class to interface :- Implements
  - interface to interface :- extends
  - interface to class :- Not possible

A class
- An interfale can extends to multiple interfaces by using extends keywords.
- An interface can't extend to class.



Ex:- public interface ISwitch {
abstract void sON();
void sOff();
}
public interface IRegulator {
void incSpeed();
void decSpeed();
}
public interface ICombo extends ISwitch, IRegulator {
}

public class FanImp implements ICombo {
@Override
void sON() {
S.O.P ("Fan Switch ON");
}
@Override
void sOff() {
S.O.P ("Fan Switch Off");
}
@Override
void incSpeed() {
S.O.P ("Fan Speed increased");
}
void decSpeed() {
S.O.P ("Fan speed decreased");
}
}

without Icombo, we need to create 2 objects
i.e ISwitch is = new Fan();
IRegulator ir = new Fan();

public class TestClass {
psvm() {
ICombo ic = new FanImp();
In order reduce the more no of object creation we declared another interface & extends to other interface

```
        ic.sOn();
        ic.sOff();
        ic.incSpeed();
        ic.decSpeed();
    }
}
```

o/p: Fan Switch ON
Fan Switch off
Fan Speed increased.
Fan Speed decreased.

- Since JDK 1.8 we can also have default concrete methods inside interface.
- Since JDK. 1.8 we can also have static concrete method inside interface.

Bcz instead of depending on subclass we can have implementation in interface since default method & static method can't be accessed outside package & class.

★ Develop a java program that calculates & displays the area of geometric shapes, specifically circles & rectangles.

- The program should define interface shape, with the method calculate area (Abstract)
- Implement 2 classes Circle & Rectangle, which should implement shape interface.
- The circle class should take radius as a parameter while the rectangle take length & width as a parameter to initialize inside a constructor
- In a main class create instance of both shapes, calculate their area using calculateArea() & display the method.

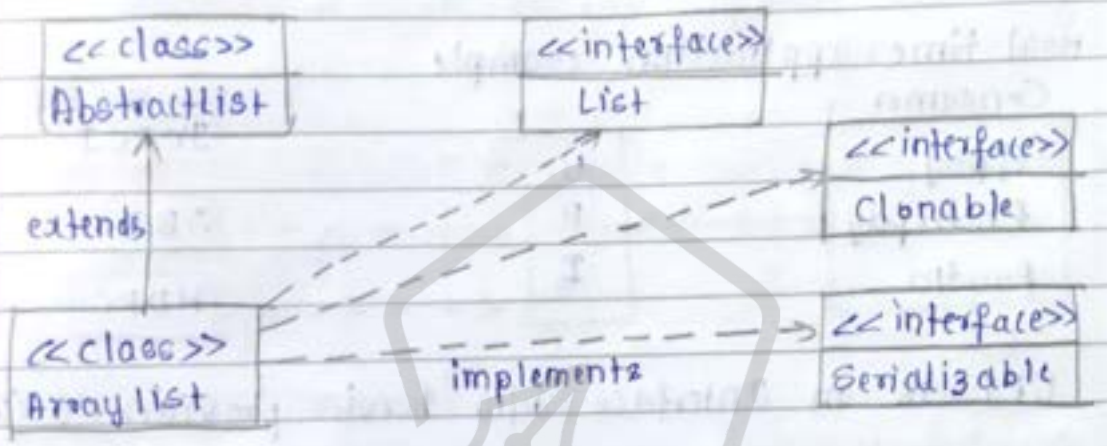
```
public interface Ishape {
    void calculateArea();
}
public class Circle implements Ishape {
    @Override
    void CalculateArea()
    {
        int r=
        double area = 3.14*r*r;
        S.O.p ("The area of circle is " + area);
    }
    int r;
    Circle ( int r) {
        this.r = r;
    }
}
public class Rectangle implements Ishape{
    int l, w;
    Rectangle ( int l, int w){
     this.l = l;
     this.w = w;
    }
    @Override
    void CalculateArea(){
        int a = l*w;
        S.O.P ("The area of rectangle :" + a);
    }
}
public class Test {
    psvm();
    Ishape i = new Circle (5);
    i.calculateArea();
```

```
Rectangle r = new Rectangle(2,3);
r.calculateArea();
}
```

05-10-2023

- ~~is it possible for a class to inherit Interface & Object class~~
- An ~~inter~~ class can extends to another class and also it can implement multiple interface.

Ex:- Real time

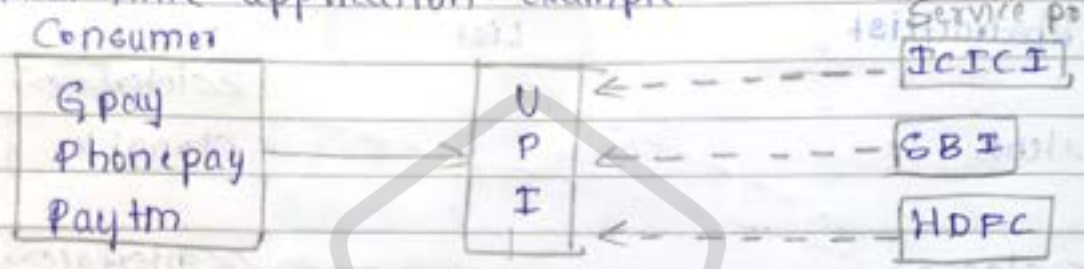



- ```
  public Interface Inf1 {
  }
  Interface Inf2 {
  }
  public class Parent {
  }
  public class Child extends Parent implements Inf1, Inf2
  {
  }
  public class child2 implements Inf1, Inf2 extends
   Parent
  {
    // syntax error
  }
  ```
  ```

- An interface cannot have instance variables and constructor
- In interface if we declare any variable by default it act as a public, static & final (Universal/Global constant).

```
interface AppConstant {
    int i = 10;   automatically / by default
                  → public static final int i = 10;
}
```

- Real time application example



UPI is an Interface b/w Service provider & Consumer

☆ Types of Interface

(I) Marker Interface
(II) Functional Interface
(III) Regular Interface.

(I) Marker Interface

- It is an empty interface which doesnot contain any methods.
- Marker interface is used to indicate JVM about the object ability/functionality.

Ex:- Serializable, RandomAccess, Cloneable

(II) Functional interface — Variable?

- It is an interface which is used to represent the main business functionality of object.
- It contains only one abstract method which represents the main business functionality.

Ex:- Comparable interface → Contains only one method. compareTo();

Comparator interface → compare();

Runnable → run(); // to run the threads

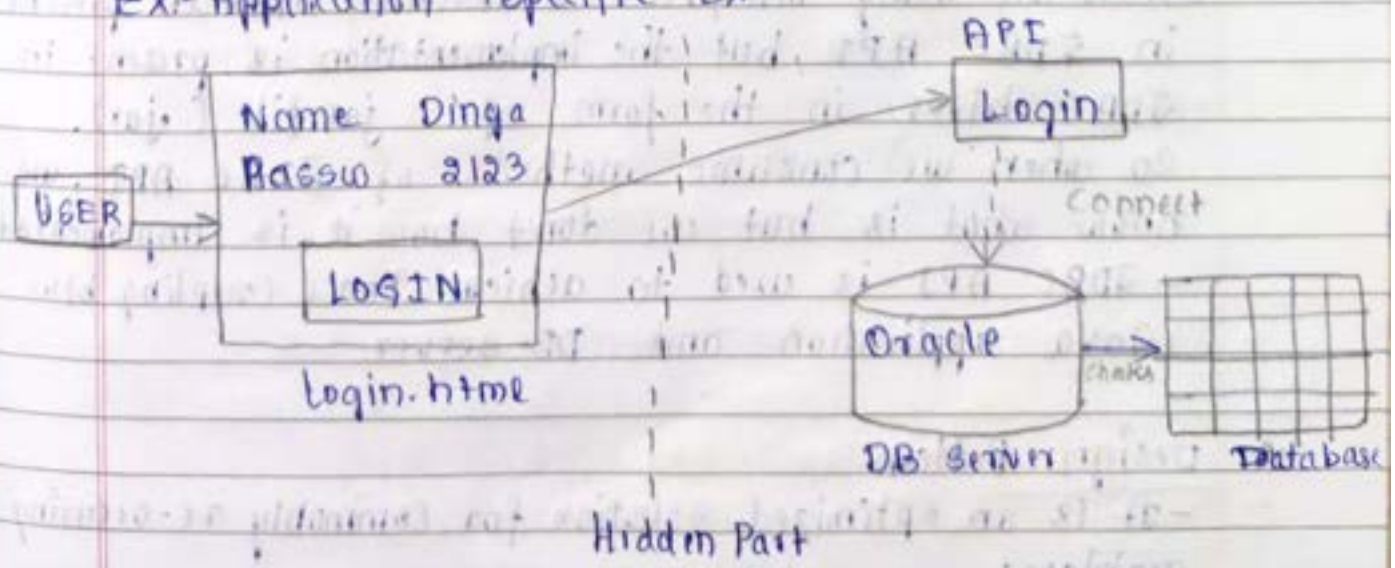
(III) Regular interface

- It is an normal interface with any no of abstract methods.

★ Abstraction (Imp★★★)

- The process of hiding the internal implementation and providing the functionalities to the users with the help of interface is called abstraction.
- Abstract class and Interface is used to achieve abstraction.

Ex:- Application specific Ex



- O/p of abstraction is Loose Coupling.

Loose Coupling :- • Change in implementation which doesnot affect the user is Loose Coupling.

Tight Coupling :- Change in implementation which affects the user is tight coupling.

30/10/2023

★ Real time example for abstraction

JDBC API is an realtime example for abstraction

- There are many interfaces & abstract method are present in JDBC API, but the implementation is present in JDBC driver in the form of jar file (.jar).
So when we consume methods of JDBC API, we know what is but we don't how it is implemented
- JDBC API is used to achieve loose coupling b/w Java application and DB server.
-

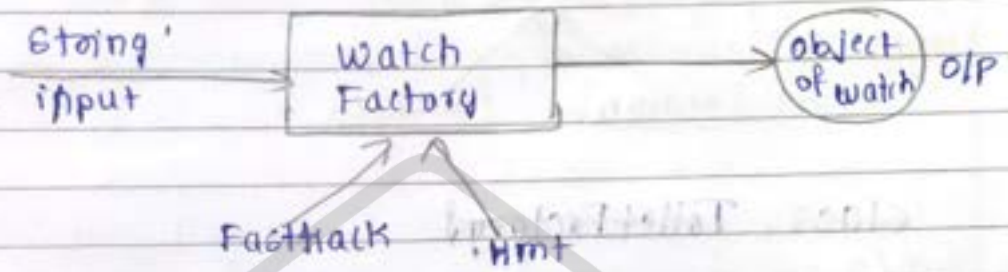
★ Design Pattern (★★) / DAO Pattern

- It is an optimized solution for commonly re-occuring problems.

+ = public
− = private
* = Protected

— There are two types of Design pattern
(i) Object creational design pattern
(ii) Factory design pattern (Imp***)

* Factory design pattern
— It is used to create & return different objects of same type.
— It always takes String as an input.

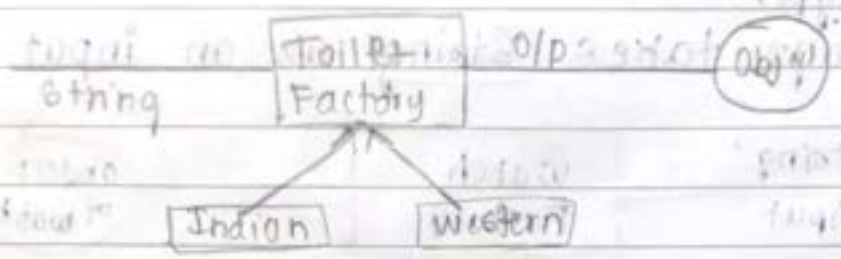


— Factory class is also called as Helper class
— Factory method is also called as Helper method.

Note :- It is always associated with 3 different types of logic (i) Implementation logic
(ii) Object Creational logic
(iii) Consumer / Utilization logic (**)

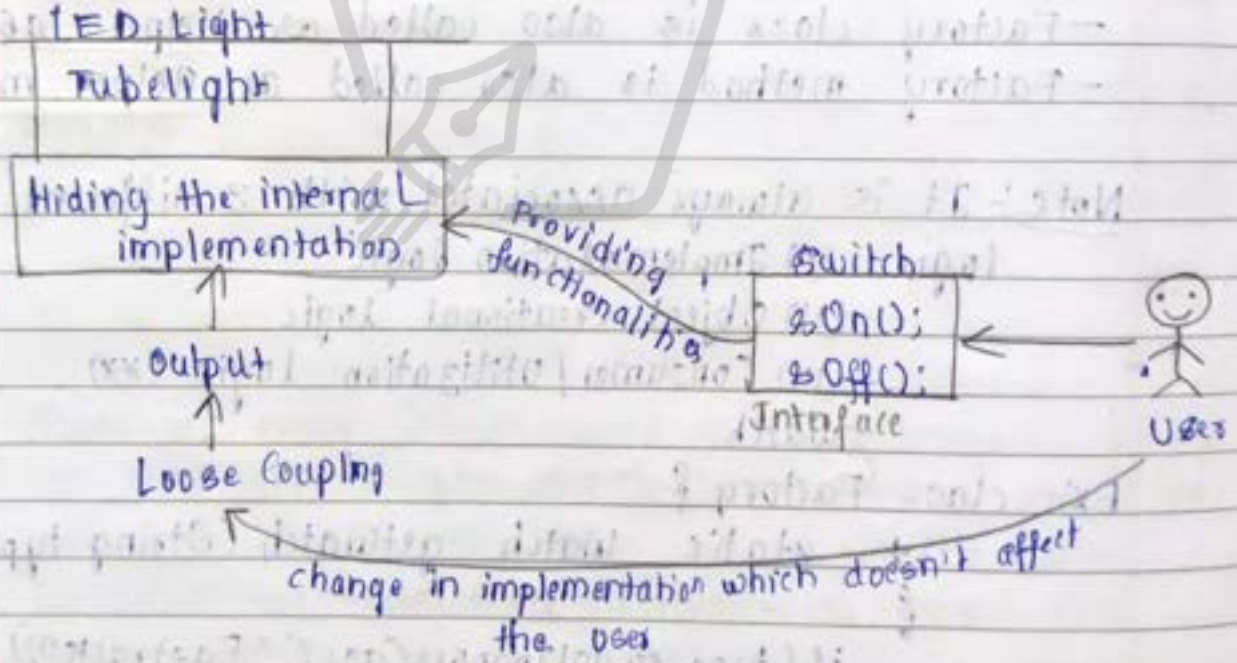
Ex:-
```
class Factory { /* Factory/ Helper class */
    + static Watch getWatch (String type)
    { /* Factory / Helper method */
        if(type.equalIgnoreCase ("Fastrack"))
        {
            return new Fastrack();
        }
        else if(type.equalIgnoreCase ("Hmt"))
        {
            return new Hmt();
        }
```

```
else
{
    S.O.P("No such watch Found");
}
}
}
```




```
class ToiletFactory{
```

Ex:- LED, Light



- Abstraction:- Hiding the internal implementation & providing the functionalities to the user with the help of interface.
- Interface:- It is an intermediate b/w user and service provider.

- Loose Coupling :- Change in implementation doesn't affect the user
- Factory design pattern always associated with 3 different types of logic (i) Implementation logic
  (ii) Object creational logic
  (iii) Consumer / Utilization logic.

(i) Implementation logic
- It is a basic fundamental logic which contains only implementation according to which an object is created.

(ii) Object Creational logic
- It is used to create an implementation object according to implementation logic. by using factory/helper method.

(iii) Consumer / Utilization logic
- It is most important logic which is used to access the functionalities from the implementation.

Note: Without writing consumer logic none of the implementation will work.

- Null Pointer Exception :- Pointing towards an object which is not present throws an exception called NullPointerException.

```
public interface ISwitch {
    void Son();
    void Soff();
}
public class Tubelight implements ISwitch {
//Implementation logic
```

```
@Override
public void sOn(){
    S.o.p("Tubelight On");
}
@Override
public void sOff(){
    S.o.p("Tubelight Off");
}
}
public class LedLight implements Iswitch{ // Implementation Logic
@Override
public void sOn(){
    S.o.p("LED Light On");
}
@Override
public void sOff(){
    S.o.p("LED light off");
}
}

//Object Creational logic
public class LightFactory {
    public static Iswitch getLight(String type){
    if(type.equalsIgnoreCase("Tubelight")
    {
        return new Tubelight();
    }
    elseif(type.equalsIgnoreCase("LED Light"))
    {
        return new LedLight();
    }
    else
```

```
    {
        S.o.p("No such light found");
        return null;
    }
}
import java.util.Scanner; // Consumer logic
public class TestLight {
    Scanner sc = new Scanner(System.in);
    psvm() {
        S.o.p("Enter the type of light");
        String type = sc.next();
        ISwitch sw = LightFactory.GetLight(type);
        if (sw != null)
        {
            sw.sON();
            sw.sOff();
        }
    }
```

03/11/2023

★ Wrapper class.

- In java for every primitive datatype there is a corresponding non-primitive datatype which we called it as wrapper class.
- All the wrapper classes are final classes, hence it cannot be inherited.
- In all the wrapper classes toString() is already overridden. Hence it prints the content rather than address.

Ex:-
```
public class Wrapper_Demo {
        psvm() {
            int i = 10; //Primitive value
            Integer wi = new Integer(120); //Non-primitive value
                                  ↓
                                  wi
```

```
Double wd = new Double(15.55); // Non primitive value
System.out.println("wi + " " + wd);
}
}
```

O/P = 12 D 15.55

Primitive Datatype	Wrapper Class (final)
byte	Byte
Short	Short
int	Integer
long	Long
double	Double
float	Float
Char	Character
boolean	Boolean

☆ Autoboxing (auto-boxing) (Imp**) Used in Collection

The process of converting primitive data type into corresponding non-primitive datatype.

Ex:-
```
public class Wrapper_Demo {
    psvm() {
        int i = 10;
        Integer wi = i;
    // Internally it creates object of Integer
    i.e Integer wi = new Integer(i);
        Double wd = 65.55;
    // Double wd = new Double(65.55);
```

☆ unboxing.

The process of automatic conversion of non-primitive data type into primitive datatype.

Ex:- public class Wrapper_Demo{
    psvm() {
        Integer wi = 120;
        Double wd = 66.6;
        Boolean wb = true;
        } Auto-boxing

        int i = new Integer(wi);
        double d = new Double(36.66);
        boolean b = new Boolean(true);
        } un-boxing
    }
}

Note:- We can convert String representation of primitive values into actual datatype by using "parse()".

★ parse()

Syntax:- + static XXX parseXXX(String s)

Ex:- + static int parseInt(String s1)
+ static double parseDouble(String s2)
+ static boolean parseBoolean(String s3)

Ex:- public class Wrapper_Demo{
    public static void main(String[] args){
        String s1 = "50";
        String s2 = "20.32";
        S.o.pln(s1+s2); // 5020.32
        int i = Integer.parseInt(s1);
        double d = Double.parseDouble(s2);
        S.o.pln(i+d); // 70.32
    }
}

- String s1 = "Hello";
  int i = Integer.parseInt(s1); ⟹ NumberFormatException
  S.O.p(i);

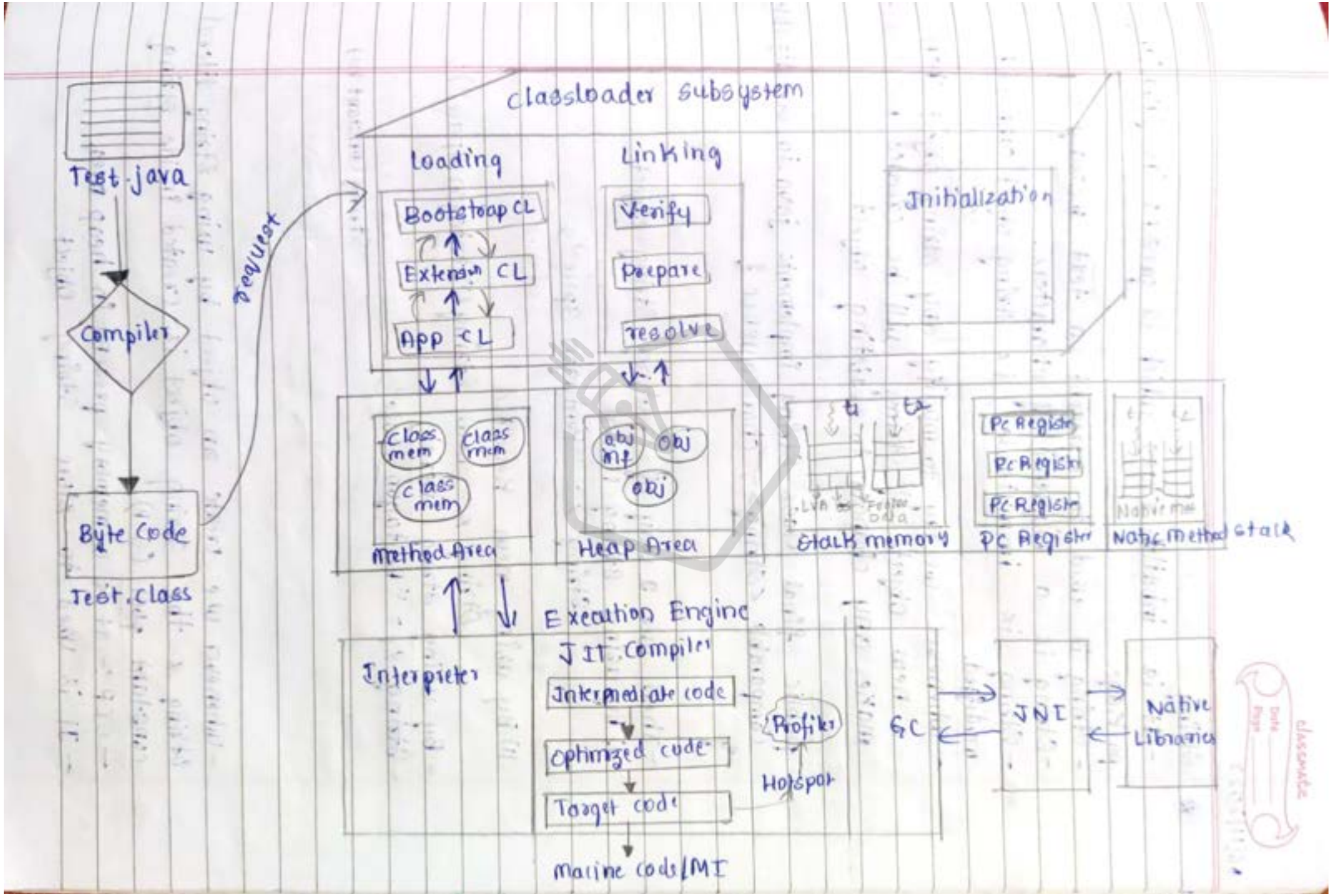
- String s1 = "true";
  boolean b = Boolean.parseBoolean(s1); ⟹ true
  S.O.p(b);

- String s1 = "Hello";
  String s2 = "Ho";
  String s3 = "50.344";
  boolean b1 = Boolean.parseBoolean(s1);
  S.O.p(b1); ⟶ false
  boolean b2 = Boolean.parseBoolean(s2);
  S.O.p(b2); ⟶ false
  boolean b3 = Boolean.parseBoolean(s3);
  S.O.p(b3); ⟶ false

04/10/2023

★ JVM Architecture

Loading:-
Loading means Reading class files and store corresponding binary data in method area.



Sharath Bhairaraju

06/11/2023

☆ String

- It is inbuilt class which is present in java.lang package.
- String is used to represent a text message
- String is a sequence of characters.
- ** String is "immutable", i.e String object cannot be modified.
- Whenever we try to modify any string object then a new object of String will be created but doesnot make any changes to existing object.

Syntax:

```
public final class String implements java.io.Serializable,
Comparable <String>, CharSequence {
}
```

- There are 2 way of creating String object:

(I) By using String literal

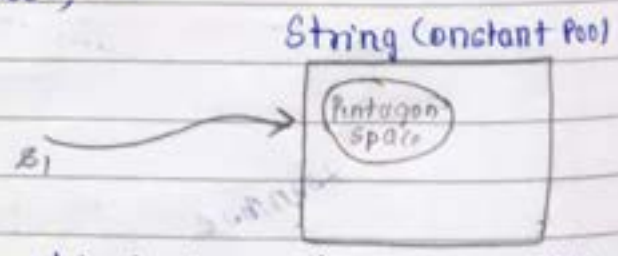
```
String s = "Pentagon Space";
```

(II) By using new Keyword

```
String s = new String("Pentagon Space");
```

- By using string literal

```
String s1 = "Pentagon Space";
```

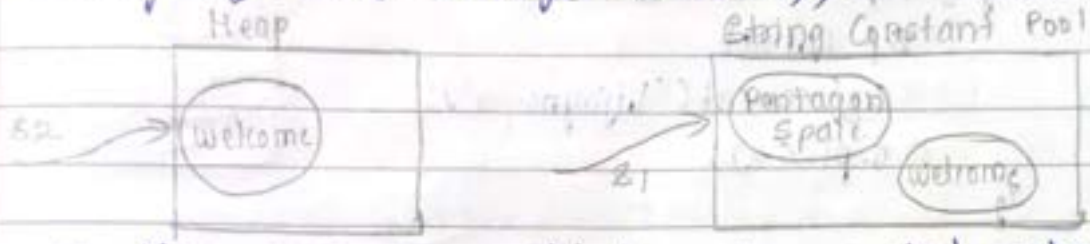


- Whenever we create an object by using String literal String s. the String object is created inside String constant pool (SCP)
- SCP :- It is a memory present in heap area.
- It is used to store String Object

- It always stores different objects of String, i.e it doesnot allow any duplicate copy.

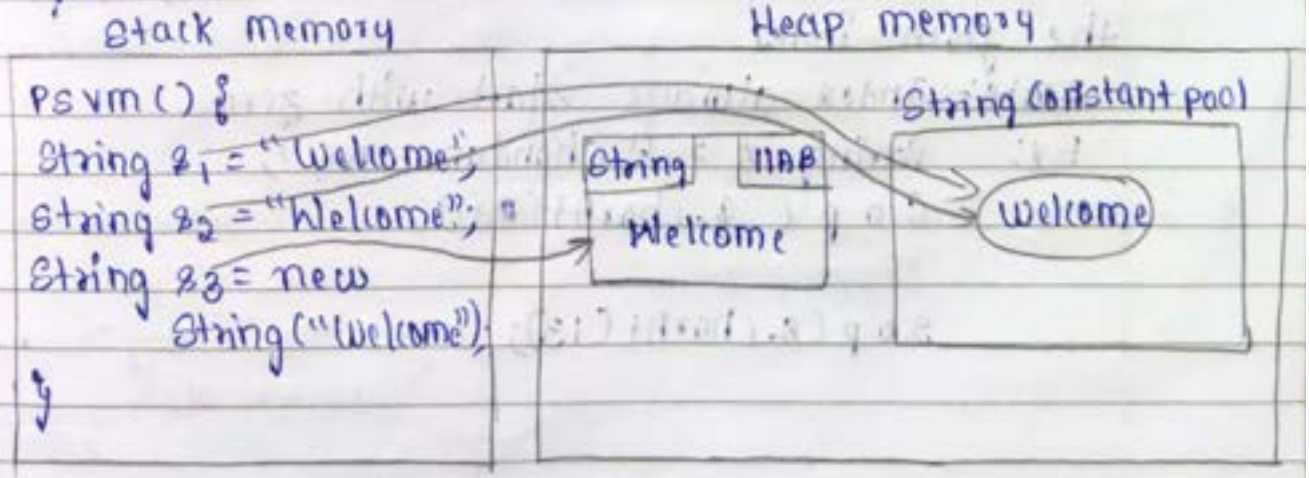
** By using new Keyword

String s2 = new String("welcome");



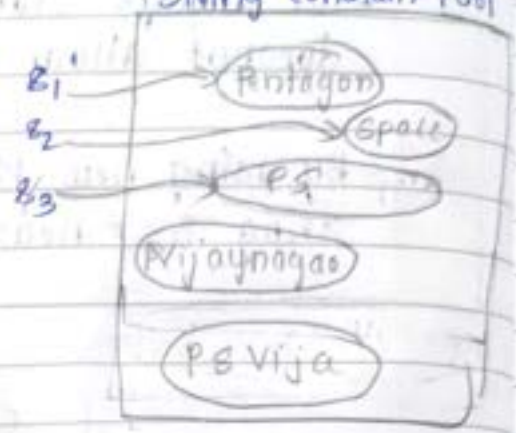
- In this case two objects gets created where one object will be created inside heap memory and one more object gets created inside String constant pool.

Ex: public class StringDemo{
    psvm() {
        String s1 = "welcome";
        String s2 = "welcome";
        String s3 = new String("welcom");
        S.o.pln(s1+" "+s2+" "+s3); //2 objects Created
    }
}



JVM .classloader	Public, String, FQCN	public static void main(String[] a)

Ex:- public class Demo{
psvm(){
String s1 = "Pentagon";
String s2 = "Space";
String s3 = s1.concat(s2);
s1.concat
s3.concat("vijaynagar");
S.o.p(s3);
}



★ Methods of String

(i) length():-
It is used to return length of given String
Ex:- String s = "Pavithra";
S.o.p(s.length());
O/p :- 08

(ii) charAt(int index):-
It is used to return the character which is present in the given index
Note:- Index always start with zero.
Ex:- String s = "Chandana NG";
S.o.p(s.charAt(03));
Output :- n
S.o.p(s.charAt(13)); // java.lang.StringIndexOutOfBoundsException

★ Write a java program to find a frequency of a character in a given String.

```
public class Frequency {
    Scanner sc = new Scanner(System.in);
    public static void main (String[] args){
        S.o.p("Enter a String");
        String s = sc.next();
        int count=0;
        for(int i=0; i<s.length(); i++)
        {
            if (s.charAt(i)==='a') {
                count++;
            }
        } S.o.p (count);
    }
```

7/11/2023

★ Immutable :- Any changes made to an object will not affect the existing object (not modified) instead of new object will get created with the changes ~~name~~ made.

Ex:- String s1 = new String("Sharath");
s1.concat("Bhairaraju");
String s2 = s1.concat("Pent");
s2.concat("Vijayanaga");
S.o.p(s1); // Sharath
S.o.p(s2); // Sharath Pent

Heap	SCP
s1 → Sharath	Sharath
Sharath Bhairaraju	Bhairaraju
s2 → Sharath Pent	Pent
Sharath Pent Vijaynagar	Vijaynagar

★ Why String is immutable?

A String object can be referred by multiple reference in this case if any one reference try to modify or change the string object then the remaining references

will get affected. Hence String Immutable.

★ String Constant Pool

It is a special memory which is present inside a heap area.

SCP always stores String object which is created by using String literal way.

★★ Why SCP?

- To increase memory Effeciency :- By reusing String literals, SCP helps to save memory. where the duplicate copy of literals cannot be stored.
- To increase the performance of application :- The most commonly used object in any application is String object. In this case ~~mul~~ if multiple String object is created for same value, then the performance will be reduced. To avoid this they introduced SCP, where only object of the same value will be created & referred by multiple reference.

Note:- String is a class which is introduced in JDK 1.0

methods

(iii). equals()

- In case of String equals() is used to compare the object content.

(iv) tolowercase()

It is used to convert the string into lowercase.

(v) trim()

It is used to remove the beginning and end space from the given string.

```
Ex:- public class Demo{
        psvm(){
        Scanner sc = new Scanner(System.in);
        S.o.p("Enter the city name");
        String type = sc.next().tolowercase().trim();
        if (city.equals("bangalore"){
            S.o.p("Welcome to garden city");
        } else if (city.equals("bidar"){
            S.o.p("Welcome to fort city");
        } else if (city.equals("mangalore")){
            S.o.p("welcome to port city");
        }
        else {
            S.o.p("Invalid city name");
        }
    }
```

* Constructors of String

(I) + String()

(II) + String(String literal)

(III) + String(String_Buffer sb)

```
Ex:- StringBuffer sb = new StringBuffer("Dinga");
     String s = new String(sb);
```

(IV) + String (StringBuilder sb)

Ex: StringBuilder sb = new StringBuilder("ding");
String s = new String(sb);

(V) + String (char[] ch)

Ex:- char[] ch = {'p', 'a', 'v', 'i'};
String s = new String(ch);
S.O.P(s); //pavi

(VI) + String (byte[] b)

Ex: byte[] b = {65, 66, 67, 68};
String s = new String(b);
S.O.P(s); //ABCD

★ StringBuffer:

- It is a class which is present in java.lang package
- It is introduced in JDK 1.0 version.
- StringBuffer is used to store sequence of characters.
- StringBuffer is "mutable", i.e any changes made to the StringBuffer object, then new object will not be created, it will modify the existing object
- when there is frequent changes needed we use StringBuffer instead of String

Ex: public class StringBufferDemo{
psvm(){
StringBuffer sb = new StringBuffer("Pavithra");
sb.append("SK");
S.O.P(sb); //PavithraSK
}
}

sb → PavithraSK

15/11/2023

SURYA Gold

Date _____ Page _____

* equals & ==

equals = Used for content comparision

== = Used for reference comparision

```
Ex:- public class Demo{
        psvm(){
            String s1 = new String("Pavi");
            String s2 = new String("Pavi");
            S.O.P(s1 == s2); // false
            StringBuffer s3 = new StringBuffer("HH");
            StringBuffer s4 = new StringBuffer("HH");
            S.O.P(s3 == s4); // false
        }
    }
```

s1 → Pavi

s2 → Pavi

```
Ex:- public class Demo{
        psvm(){
            String s1 = new String("Pavi");
            String s2 = new String("Pavi");
            S.O.P(s1.equals(s2)); // true
            StringBuffer s3 = new StringBuffer("Pavi");
            StringBuffer s4 = new StringBuffer("Pavi");
            S.O.P(s3.equals(s4)); // false
        }
    }
```

- In case of String class equals() is meant for content comparision.
- In case of StringBuffer equals() is meant for reference comparision.
- The methods of StringBuffer are synchronised.

Synchronized

- If the method is synchronized only one thread can able access at a time but it doesnot allow for multiple threads.

Ex:- Bus, Movie ticket booking

- It reduces the performance of application since another thread needs to wait for the completion of execution.

★ Constructor of StringBuffer

(I) public StringBuffer();
(II) + StringBuffer(int initialCapacity);
(III) + StringBuffer(String str)
(IV) + StringBuffer(CharSequence ch)

The initialCapacity of StringBuffer is 16 and the incremental capacity is $\boxed{(CC+1)*2}$

CC = Current capacity / Initial Capacity

Ex:- Capacity

```
+ class Capacity{
    psvm(){
    StringBuffer sb1 = new StringBuffer();
    sb1.append("PavithraSKsampig");
    S.O.P("Initial Capacity:" + sb1.capacity());//16
    sb1.append("Hosahalli");
    S.O.P("Incremental Capacity:" + sb1.capacity());//34
    StringBuffer sb2 = new StringBuffer(100);
    sb2.append("Pavi");
    S.O.P("capacity" + sb2.capacity());//100
```

Differences

Imp* *

String	String Buffer	String Builder
- Immutable	- Mutable	- objects are mutable
- Have concat()	- Have append()	- Have append()
- equals() for Content comparision	- equals() for reference or address comparision	- equals() for address/ reference comparision
-	- Every methods are Synchronized	- No methods are Synchronized
- It is thread safe (All immutabe obj bydefault thread safe because no one can change value	- At a time only one thread is allow to operate of SB object & Hence it is thread safe	- At a time multiple thread are allowed to operate on String Builder object and hence it is not thread safe.
	- Threads are required to wait to operate on SB object and hence relatively the performance is slow	- Threads are not required to wait to operate on String Builder object & hence relatively the performance is high

- String s1 = new String("Pavi");
  StringBuffer s2 = new StringBuffer(s1);
  S.o.p(s2.capacity()); // 20 (initial + 16)
  (Capacity 20)

☆ String — String Buffer — String Builder

- Immutable/ objects are not changed
- objects are

Sharath Bhairaraju

SURYA Gold

Date _____ Page _____

☆ Except String class any other class in java is Immutable?

Yes, All the wrapper class are immutable class in java.

☆ Can we create a immutable class

Yes

```
public class Test {
    int i;
    Test (int i)
    {
        this.i = i;
    }
    public Test modify (int i)
    {
        if (this.i == i)
        {
            return this ;
        }
        else
            return new Test(i);
    }
}
class Demo
{ psvm() {
    Test t1 = new Test (100);
    S.o.p (t1);
    Test t2 = t1.modify (101); //different address
    s.o.p (t2);                 will print
  }
}
```

SURYA Gold

Date _____ Page _____

07/11/2023

☆ Array

- It is a collection of similar data (Homogenious data)
- It is used to store and retrieve multiple values
- Array is fixed in size

Syntax:

```
datatype[] var_name;
datatype[] var_name = new datatype[size];
```

Note:- Array can be used to store primitive data and non-primitive data

Ex: int a1[] = {20, 40, 10, 15, 60};

i =	0	1	2	3	4
a1:	20	40	10	15	60

```
//index of array starts with 0 & ends with (n-1) or a1.length() -1
S.o.p(a1[0]);
S.o.p(a1[1]);
S.o.p(a1[0]);
S.o.p(a1[3])
S.o.p(a1[4]);
S.o.p(a1[5]); // ArrayIndexOutofBound Exception
/* to display array elements using loops */
for (int i=0; i<a1.length(); i++)
{
    S.o.p(a[i]);
}
// a1.length(); is 5
```

Effecient way of retrieving

Ex:- To insert an elements in Array

```
int[] arr = new int[5];
/* for loop for inserting an element */
for(int i=0; i<arr.length; i++)
```

```
{
    arr[i] = sc.nextInt();
    S.o.p("Array elem" arr[i]);
}
```

i = 0 ; 0 < 5	i = 1 ; 1 < 5	i = 2 ; 2 < 5	i = 3 ; 3 < 5
arr[0] = 10	arr[1] = 20	arr[2] = 30	arr[3] = 40
i = 4 ; 4 < 5	i = 5 ; 5 < 5		
arr[4] = 50	for loop		

☆ write the java program to find frequency of given element in array.

```
public class Frequency{
    psvm(){
     scanner sc = new Scanner(System.in);
     S.o.p("Enter size of an array");
     int size = sc.nextInt();
     int a[] = new Int[size];
     for(int i=0; i<a.length; i++){
        a[i] = sc.nextInt();
        S.o.p(a[i]+" ");
     }
     S.o.p("Enter the element");
     int a1 = sc.nextInt();
     for(int i=0; i<a.length; i++)
     {
        if(a[i] == a1) {
            count++;
        }
     }
     S.o.p("Frequency of element :" + count);
    }
}
```

10/11/2023

* Write a program to print the biggest element from the array

```
public class Frequency{
    psvm(){
        int[] a = {10, 20, 30, 40};
        int max = 0;
        for(int i=0; i<a.length; i++){
            if (a[i] > max){
                max = a[i];
            }
        }
        S.O.P("Biggest element in array is " + max);
    }
}
```

Tracing:

1st iteration.

• i = 0 , 0 < 4 , T

a[0] > 0

10 > 0 (T)

Then $\boxed{max = 10}$

i++ ⇒ 01

2nd iteration

• i = 1 , 1 < 4 , T

a[1] > max

20 > 10 (T)

$\boxed{max = 20}$

i++ ⇒ 02

3rd iteration

• i = 2 , 2 < 4 , T

a[2] > max , T

30 > 20

$\boxed{max = 30}$

i++ ⇒ 3

4th iteration

• i = 3 , 3 < 4 , T

a[3] > max

40 > 30 , T

$\boxed{max = 40}$

i++ ⇒ 4

5th

• i = 4 , 4 < 4 False

Then

$\boxed{max = 40}$

☆ Write a java program to sort an array in descending order

```
public class Ordering{
    psvm(){
        int a[] = {45, 10, 60, 25};
        int size = a.length; int temp=0;
        for(int i=0; i<size; i++) {
            for(int j= i+1; j<size; j++) {
                if (arr[i] < arr[j]){
                    temp = arr[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }
        S.O.p("Elements after sorting" + a[i]);
    }
}
```

Tracing:

size = 04

45	10	60	25
0	1	2	3

- For int i=0; 0<4; j↑
  - j=0+1; 1<4
    a[0] < a[1]
    45 < 10 False
  - j=2; 2<4
    a[0] < a[2]
    45 < 10 False
  - j=3; 3<4
    a[0] < a[3]
    45 < 60 True

temp = a[1] = 45;
a[1] = a[2] = 60;
a[2] = temp = 45;
Then. [60 | 10 | 45 | 25]
↑i

- j=4 ; 4<4 False
i++ ;

- i=1 ; 1<4 ; True [60 | 10 | 45 | 25]
↓i ↑j
- j=i+1=2 ; 2<4 ; True
a[1] < a[2]
10 < 45 True
temp = 10 ;
a[1] = 45;
a[2] = 10 ;

Then [60 | 45 | 10 | 25]
↓i
- j = 3; 3 <4; T
a[1] < a[3]
45 < 25 F
i++;

- i=2; 2<4; T [60 | 45 | 10 | 25]
0 1 2 3
↓i ↑j
- j=3; 3<4 T
a[2] < a[3]
10 < 25 T
temp = 10
a[2] = 25
a[3] = 10

Then [60 | 45 | 25 | 10]
↑i

- j=4; 4<4 False
i++;

- i=3; 3<4; T [60 | 45 | 25 | 10]
↑i

- j = 4 ; 4 < 4 , False
  i++

- i = 4 ; 4 < 4 , False

The Elements after Sorting 60 45 25 10

Note :- In array we don't have any inbuilt method to deal with data

Array can store Non-primitive data

Ex:-
```
public class Customer {
        String cname;
        long contact;
        Customer( String name, long con){
            cname = name;
            contact = con;
        }
}
class Bank {
        psvm() {
            Customer c1 = new Customer("pa",123);
            Customer c2 = new Customer("ch", 346);
            Customer c3 = new Customer("sa",987);
            Customer c[] = {c1, c2, c3};
            for(int i=0; i<c.length; i++){
                S.O.P( c[i]);
            }
        }
}
```

O/p :- Adress of c_1 c_2 c_3 will be printed
to print the value we have to override to string() in Customer class

Then, public class Customer{

```
    String cname;
    Long contact;
    Customer(String name, long con){
        cname = name;
        contact = con;
    }
    @Override
    public String toString(){
        return cname+" "+contact;
    }
}
class Bank{
    psvm(){
        Customer c1 = new Customer("pavi", 12345);
        Customer c2 = new Customer("chandu", 4567);
        Customer c3 = new Customer("Spandu", 93537);
        Customer c[] = {c1, c2, c3};
        for(int i=0; i<c.length; i++){
            S.O.P(c[i]);
        }
    }
}
```

O/p: Pavi 12345
Chandu 4567
Spandu 93537

11/10/2023

★ Drawbacks of Arrays

- It is fixed in size.
- There is no inbuilt methods in arrays.

utility methods = Common methods like add(), remove()
Dynamic/Customized program = Using Scanner

- It stores only homogenous data.
- Array stores primitive data, in case of non-primitive it stores referrals.
- Arrays doesnot have any generic
- Array doesnot maintain any datastructure

☆ Framework
- A software framework is a abstraction which contains generic functionalities which can be extended and used and also customizable by additional user written code
- A framework may contain many inbuilt classes, interfaces, & readymade utility methods.
- A framework is reusable
- Mainly used for application development.
- They are tested
- Uses best practices & necessary design patterns
- Different frameworks are available for different kinds of application development.
- A framework is used to develop many s/w application products.
  Ex:- Springs, Hibernate, Angular, react, nodejs etc

☆ Collection Framework
- It deals with the collection/group of data.
- It has pre-defined interfaces, implementation classes, utility methods which are collectively present in java.util package.

SURYA Gold

Date _____ Page _____

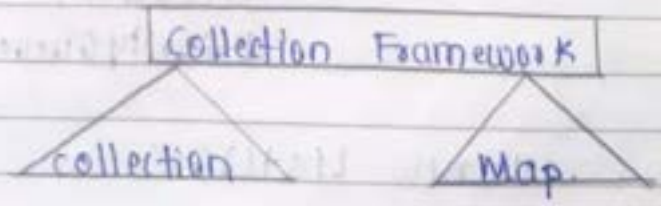
☆ Why Collection Frameworks?

- To store heterogenous data (data of different types).
- Effective way for data handling.
- Defined datastructures and ready made methods

☆ Difference Between Array & Collection (Imp**)

Arrays	Collection
- Homogenous in nature wherein only one type of data can be stored	- Heterogenous in nature wherein multiple types of data can be stored.
- It is static or fixed in size	- It is dynamic or elastic
- Doesnot have inbullt methods for data handling	- Has many inbuilt methods for data handling
- Can store primitive data	- Cannot store primitive data.
- Doesnot use any data Structure	- Uses data structures.
- In case of non-primitive data array can store only reference.	- It can store object as well as reference.
- Arrays doesnot have any generic	- Collection contains generic

☆ Collection Hierarchy



Note:- Technically collection is interface which is present in java.util package, Since JDK 1.2V

☆ To create implementation object of Collection

		Implementation class
–	Collection col = new	ArrayList();
		Vector();
		LinkedList();
		Stack();
		HashSet();
		LinkedHashSet();
		TreeSet();
		PriorityQueue();

– Collection col = new List(); // error List is interface

SURYA Gold
Date _____ Page _____

we can't create object for interface.
- we use add method to insert the object into collection i.e coll.add(10);

Note:- In collection whenever we try to store primitive data by default primitive values will be auto-box and stores the object of primitive values with the help of wrapper classes.

```
Ex:- public class CollDemo{
        psvm(){
            Collection c = new ArrayList();
//to create Collection, we need to import from java.lang.*;
            c.add(20); internally new Integer(20);
            c.add(20, 55);
            c.add('A');
            c.add("Sharath");
            c.add(true);
            System.out.Println(c);
            //20 : 20:55: A Sharath true
        }
}
```

5-11-2023

★ Generic
- It is a feature introduced in JDK 1.5
- It is used to specify the type of element
  Syntax:- Generic <E>

★ Collection with generic
Ex:- • Collection<E> col = new ArrayList<E>();
• Collection<Integer> col = new ArrayList<Integer>();
• Collection<String> col = new LinkedList<String>();

Collection is designed for object, Here add() in collection takes element/object as parameter
So, we can't store primitive data

SURYA Gold
Date _____ Page _____

- Collection<Customer> col = new Vector<Customer>();
- Collection<int> col = new Vector<int>(); //error

Reg collections doesnot store primitive data

Ex:-
```
import java.util.ArrayList;
import java.util.Collection;
public class CollDemo {
    psvm() {
        Collection<String> c = new ArrayList<String>();
        c.add("Pavi");
        c.add("Chandhu");
        c.add("Kruthi");
        c.add("Spandhu");
        c.add(120); // error
        S.O.P(c);
    }
}
```
O/p:- Pavi Chandhu Kruthi Spandhu

Ex:- Collection without generic

```
import java.util.*;
public class Coll {
    psvm() {
        Collection c = new ArrayList();
        c.add("Pavi");
        c.add(101);
        c.add(15.08);
        c.add(true);
        c.add('C');
        S.O.P(c);
    }
}
```
O/p:

Note:- Whenever we need to store similar type of elements then we can use generic (Type Safety).

☆ Methods of Collection (Imp***)

(i) int size();
(ii) boolean isEmpty();
(iii) boolean contains(Object o);
(iv) Object[] toArray();
(v) boolean add(E e);
(vi) boolean remove(Object o);
(vii) boolean containsAll(Collection c);
(viii) boolean addAll(Collection c);
(ix) boolean removeAll(Collection c);
(x) boolean retainAll(Collection c); // retains particular c & remove others
(xi) void clear();
(xii) Iterator<E> iterator();

Ex:- For methods

```
import java.util.*;
public class Demo{
    psvm(){
        Collection<String> cs = new ArrayList<>();
        cs.add("Pavi");
        cs.add("Moni");
        cs.add("Niki");
        cs.add("Shalini");
        S.o.p(cs); //Display the element of cs
        Collection<String> ec = new ArrayList<>();
        ec.add("Male");
        ec.add("Spandhu");
        ec.add("manoj");
```

```
S.o.p(ec);
Collection<String> me = new LinkedList<>();
me.add("Santhu");
me.add("Rohan");
me.add("Varun");
S.o.p(me); // Santh, Rohan, Varun
Collection<String> google = new LinkedList<>();
google.addAll(cs);
google.addAll(ec);
google.addAll(me);
S.o.p(google);
S.o.p(google.size()); //10
S.o.p(google.contains("Moni")); //true
S.o.p(google.remove("manoj")); //true
S.o.p(google.containsAll(cs)); //true
S.o.p(goole.removeAll(me)); //true
S.o.p(google.containsAll(ec)); //false (manoj is removed)
google.retainAll(cs);
S.o.p(google.size()); //4
google.clear();
S.o.p(google.size()); //0
S.o.p(google.isEmpty()); //true
}
}
```

-11-2023

☆ List

- List is an subinterface of collection interface presents since JDK 1.2V
- List is an index based.

Sharath Bhairaraju

- List can store duplicate elements
- List can store null values
- List maintains insertion order

☆ Methods of List interface (List Specific methods)

(I) void add(int index, E element);
(II) boolean addAll(int index, Collection col);
(III) E get(int index);
(IV) Int indexOf(Object o);
(V) Int lastIndexOf(Object o);
(VI) ListIterator<E> iterator();
(VII) ListIterator<E> listIterator();
(VIII) ListIterator<E> listIterator(int index);
(IX) E remove(int index);
(X) E set(int index, E element); // Replace the element at given index
(XI) List<E> subList(int fromIndex, int toIndex);

```
Ex:- public class ListDemo{
        psvm(){
            List<String> ls = new ArrayList<>();
            ls.add("Dhoni");
            ls.add("Virat");
            ls.add("Yuvraj");
            ls.add("Virat");
            ls.add(null);
            S.O.P(ls);
        }
}
```

O/P:- [Dhoni, Virat, Yuvraj, Virat, null]

SURYA Gold

Date _____ Page _____

- List can be Iterated in 04 different ways:
  (i) Using for loop
  (ii) Using for-each loop (enhanced for loop),
  (iii) Using Iterator and ListIterator (iv)

Ex:
```
public class ListIteration{
    psvm() {
        List<String> ls = new ArrayList<>();
        ls.add("Thouseef");
        ls.add("Pallavi");
        ls.add("Sanjay");
        S.o.pln(ls);
        S.o.pln("Using for loop");
        for(int i=0; i<ls.size(); i++){
            S.o.p(ls.get(i));
        }
        S.o.p("Iterate using for each loop")
        for (String var: ls)
        {
            S.o.p(var);
        }
    }
}
```

o/p:- [Thouseef, Pallavi, Sanjay]
Using for loop
Thouseef
Pallavi
Sanjay
Iterate using for each loop
Thouseef
Pallavi
Sanjay

Arraylist

- It is an implementation class of List interface present since JDK 1.2V
- It inherits all the features of List interface
- It is index based.
- It stores duplicate values
- It allows null values
- It maintains insertion order.
- Arraylist internally stores the data in the form of array with initial capacity 10.
- The incremental capacity of Arraylist is depends on formula :

$$\boxed{\text{incremental capacity} = \frac{CC \times 3}{2} + 1}$$

- Arraylist maintains an internal datastructure called growable array or resizeable array.
- It implements marker interface like Serializable, Cloneable, RandomAccess.
- There are 3 overloaded constructor present in Arraylist.
  (i) Arraylist( )
  (ii) Arraylist(int initialCapacity)
  (iii) Arraylist(Collection c)

Advantages (Situation to prefer) Arraylist

- It internally stores the data in form of array. Hence the memory is continous, which is a best choice for retrival & search operation.
- Arraylist is a best choice for retrieval & search operation. Since it is an index based the time taken to search / retrieve any element will be same

- Disadvantages [Situation not to Prefer] Arraylist
  - If there is frequent modification in our collection then an Arraylist is not a good choice.
  - whenever we add/remove the elements in between Arraylist, each elements will be shifted which performs shift operation.
  - Because of shift operation the performance of an application will be decreased/reduced.

a1	10	40	20	15
	0	1	2	3

a1.add(5, 2)

Then

a1	10	40	5	20	15
			2	3	4

a1.remove(2)

a1	10	5	20	15
	0	1	2	3

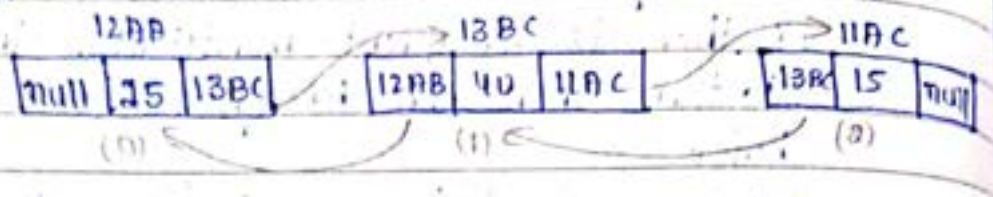
18/11/2023

★ LinkedList:
- It is an implementation class of List interface present since JDK 1.2V
- It stores the data in the form of nodes.
- Each node will store previous node address, next node address and current node address

Node → | Prev | Current data | Next |

- In case of 1st node it doesnot have previous node address
- In case of Last node it doesnot have next node address.
- Each nodes are connected to each other.

Ex:

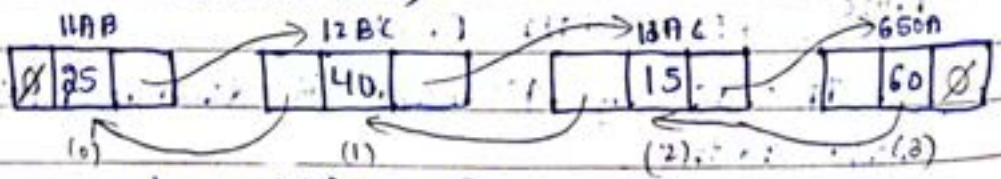


- LinkedList implements marker interface like Serializable, cloneable but not RandomAccess
- The initial capacity & incremental capacity is not applicable for LinkedList.
- There are two overloaded constructor for linked list
  (I) public LinkedList()
  (II) public LinkedList(Collection C)
- The internal data structure maintained by LinkedList is **Doubly LinkedList**.

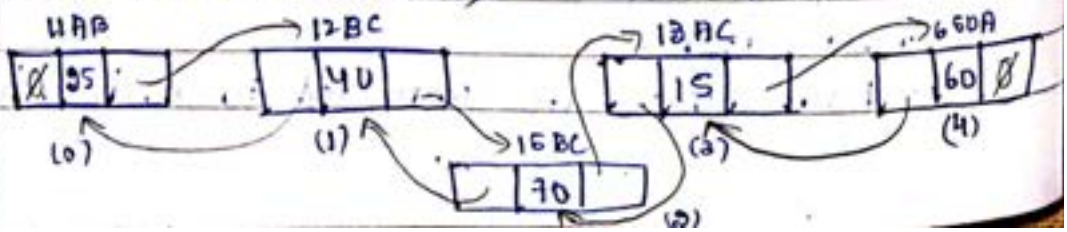
- **Situations to use LinkedList**

Whenever we have frequent modification of data in a collection, then we have to prefer LinkedList, Since it doesnot have any shift operation

```
Ex:- LinkedList<Integer> ls = new LinkedList<>();
     ls.add(25);
     ls.add(40);
     ls.add(15);
     ls.add(60);
```




```
ls.add(2, 70)
```



SURYA Gold
Date _____ Page _____

• Situations not to use LinkedList

Whenever we have search & retrieval operation on a collection then it is not a good choice to prefer linked list. Since whenever we retrieve any element it starts searching from first node.

Note:- Compared to Vector and ArrayList, LinkedList consumes more memory bcz each node going to stores the address of previous node, next node and current node.

Differences b/w ArrayList, LinkedList, Vector

★ Vector

- It is an implementation class of List Interface present since 1.0v
- Vector is legacy class (outdated class)
- Vector is single threaded class.
- The initial capacity of vector is 10 and the incremental capacity is double the current capacity (2*(C))
- In case of vector we can control the growth of capacity by using overloaded constructor.
- There are 4 overloaded constructor in vector.
  (I) public Vector()
  (II) public Vector(int initialCapacity)
  ★★ (III) public Vector(int initialCapacity, int capacityIncrement)
  (IV) public Vector(Collection C)
- Since Vector is legacy class, all the methods of vector are synchronized.

- Since all the methods are synchronised, Vector is thread safe.
- It implements the marker interface like Serialize, Cloneable, RandomAccess.
- It stores the data in form of Array.
- Vector is similar to ArrayList but with few differences

(I) Vector contains synchronized method.
(II) It contains many legacy methods that are not part of collection framework
(III) In vector we can control growth of incremental capacity.
(IV) Vector has 04 overloaded constructor & ArrayList has 3 overloaded Constructor
(V) Vector is thread safe.

* Whether vector is faster in performance | ArrayList
ArrayList, Because vector contains synchronized method, Only one thread can access the object at a time. Hence vector performance is less compared to the ArraList.

30/11/2023

* Arraylist

(I) Write a java program that contains an Employee class with the following properties;
eid (int), ename (String), Department (Department)

(II) Now create another class named Department which contains the properties
did (int), dname (String), designation (String)
(III) Now create another class named EmployeeApp

which contains the main method.

(iv) Now write code that prompt the user to enter the Employee details like id, name and prompt Department details like department id, name, disignation details in Arraylist object.

(vi) Enter atleast 3 employees & department info print those details to the output.

```
public class Employee{
    int eid;
    String ename;
    Department dept;
    public Employee(int id, String name,
                Department department) {
        eid = id;
        ename = name;
        dept = department;
    }
    @Override
    public String toString(){
        return this.ename+" "+this.eid+" "+
                this.dept;
    }
}

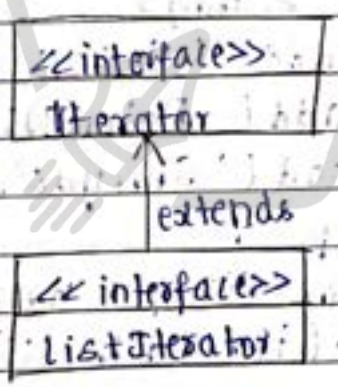
public class Department{
    int did;
    String dname;
    String designation;
    public Department(int id, String name, String role)
```

```
{
    did = id;
    dname = name;
    designation = role;
}
@Override
public String toString() {
    return this.did + " " + this.dname + " "
            this.designation;
}
}
import java.util.*;
public class EmployeApp{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        ArrayList<Employee> eCol = new ArrayList<
                                    ();
        for(int i=0; i<2; i++){
            S.o.p("Enter a emp & dept details");
            S.o.p("Enter emp id");
            int eid = sc.nextInt();
            S.o.p("Enter emp name");
            String ename = sc.next();
            S.o.p("Enter emp department id");
            int did = sc.nextInt();
            S.o.P("Enter emp department name");
            String dname = sc.next();
            S.o.p("Enter emp designation");
            String designation = sc.next();
            Department d = new Department(did,
                        dname, designation);
            Employee e = new Employee(eid, ename,
                                    dept);
```

```
    eColl.add(e);
  }
  for (Employee var : eColl)
  {
      S.O.p(var);
  }
}
}
```

☆ Iterator and ListIterator (Imp***)

- Iterator & ListIterator are interface which is present in java.util package



- ListIterator is a sub-interface of Iterator
- The most common way of iterating collection is by using Iterator or ListIterator.

- Iterator
  - Iterator is not an index based.
  - Since it is not an index based we can use Iterator to iterate entire collection.
  - Iterator is used to interate only in forward direction (uni-directional)

- There are 3 methods in Iterator
(i) + boolean hasNext();
(ii) + E next();
(iii) + void remove();

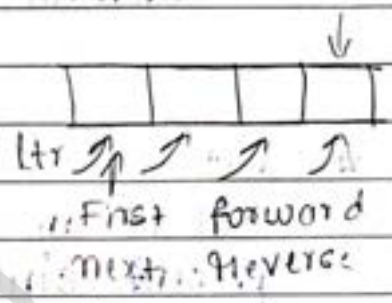
- Iterator can be used to iterator "MAP"
- To create object for Iterator
  Create Any Impl obj of collection Interface
  ↓
  Iterator<E> itr = var.name.iterator();

```
Ex:- public class Demo {
        public static void main(S[] args){
            Collection<String> col = new ArrayList<>();
            col.add("Virat");
            col.add("Rahul");
            col.add("Shreyas");
            S.o.p(col); [Virat, Rahul, Shreyas]
            Iterator<String> itr = col.iterator();
            while(itr.hasNext()) {
                String name = itr.next();
                S.o.p(name); Virat, Rahul, Shreyas
                if(name.equals("Rahul"))
                {
                    itr.remove();
                }
            }
            S.o.p(col):- [Virat, Shreyas]
        }
    }
```

21/11/2023

- **ListIterator**
  - It is an sub-interface of Iterator interface
  - It is an index based, hence it is used to iterate only List but not Set & queue.
  - ListIterator is used to iterate in forward and reverse direction. (bi-directional)
  - It inherits the methods of Iterator. along with it has few more methods

(i) boolean hasPrevious();
(ii) E previous();
(iii) int nextIndex();
(iv) int previousIndex();
(v) void set(E e);



- To create object for iterator

```
List<E> ls = new LinkedList<>();
ListIterator<E> itr = ls.listIterator();
```

Ex:-

```
import java.util.*;
public class ListItDemo{
    psvm(){
        List<String> ls = new ArrayList<>();
            ls.add("Sanjay");
            ls.add("Harini");
            ls.add("Sharath");
        ListIterator<String> itr = ls.listIterator();
        S.o.p("Iterating in forward direction\n");
        while(itr.hasNext())
        {
            S.o.p(itr.next());
        }
        S.o.p("\nIterating in reverse direction\n");
```

Difference b/w Iterator & ListIterator

```
while (itr.hasPrevious()){
        S.o.p( itr.previous() + " ");
}
}
}
```

o/p:- Iterating in forward direction
Sanjay Harini Sharath
Iterating in reverse direction
Sharath Harini Sanjay

Set

- It is an sub-interface of Collection interface present since JDK 1.2v.
- Set doesnot store any duplicate value
- Set is used to maintain uniqueness.
- Set is not an index-based.
- Set can be created using 3 implementation object

```
                TreeSet<>();
Set<E> s = new  HashSet<>();
                LinkedHashSet<>();
```

HashCode

- It is an integer representation of an object
- We can get the hashCode of an object by using hashCode().
- hashcode() is present in supermost class called object class.

Ex:-
```
public class Emp{
        psvm(){
            Emp e = new Emp();
```

```
System.out.println(e.hashCode());//118352462
System.out.printf("%x", e.hashCode());
}                                  // 70deaHe
}                                  (Hexa decimal)
```

- hashCode() is overridden in all the wrapper class & also in String class based on HashCode contract
- Return type of hashCode() is int
- HashCode values can also be negative

Ex:
```
public class HashCodeDemo{
    pvm(){
        String s1 = new String("Pavi");
        S.o.p(s1.hashCode());// 69162543
        Double d = new Double(25.56);
        S.o.p(d.hashCode());// 1758219
        Integer i = 45;
        S.o.p(i.hashCode());//45
    }
}
```

Iterator	List Iterator
- It is an interface present in java.util package.	- It is sub-interface of Iterator.
- It is not index based.	- It is index based.
- It iterates entire collection	- It is used to iterate only List not Set & Queue
- It iterates only in forward direction	- It iterates in both forward & reverse direction

20/11/2023

☆ HashCode Contract

- Two different objects of same type with same content must have same hashcode.
- Two different object of same type with different content must have different hashcode
- Two different object of different type must have different hashcode.

Ex:-

```
public class Demo{
    psvm () {
        String s1 = new String("Guldu");
        String s2 = new String("Guldu");
        - S.o.p (s1.hashcode() + " == " + s2.hashcode());
        Double d1 = new Double (25.68);
        Double d2 = new Double (30.65);
        - S.o.p(d1.hashcode() + " != " + d2.hashcode());
        - S.o.p (s1.hashcode() + " != " + d1.hashcode());
    }
}
```

O/P:- 69762543 == 69762543

1758219731 != 1758416389

697662543 != 1758219731

☆ Hash Set

- It is the implementation class of subinterface Set. present since JDK 1.2V.
- It is an hash based collection, where as hashcode of object will be stored
- Hashset doesnot maintain any insertion order, but the insertion order is depends on the hashcode of an Object.

- Hashset internally maintains a data structure called HashTable (It allow only one null value
- The initial capacity of Hashset is 16 and the filled ratio is 0.75 (75%)
  ↳ or Loadfactor
- there are 4 overloaded constructors in Hashset
  (i) + Hashset() { }
  (ii) + Hashset(Collection c)
  (iii) + Hashset(int initialCapacity, float loadfactor)
  (iv) + Hashset(int initialCapacity)
- Hashset implements marker Interface like cloneable, Serilizable but not Random Access.

Ex:-
```
public class Demo{
    psvm(){
        Set<String> hs = new Hashset<>();
        hs.add("Pavi");
        hs.add("Chandhu");
        hs.add("null");
        hs.add("Chandhu");
        hs.add("null");
        hs.add("Spandhu");
        System.out.println(hs);
    }
}
```
o/p:- [null, Pavi, Spandhu, Chandhu]

★ LinkedHashSet
- It is the implementation class of subinterface Set present since JDK 1.4v.
- It maintains insertion order.

- The Internal data structure maintained by LinkedHashSet is Hash table & Linked List
- The initial capacity is : 16 & load factor is 0.75 (75%)
- There are 4 overloaded Constructor :
  (i) + LinkedHashSet()
  (ii) + LinkedHashSet(int InitialCapacity)
  (iii) + LinkedHashSet(int InitialCapacity, float LoadFactor)
  (iv) + LinkedHashSet(Collection c)
- It implements marker interface like Cloneable, Serilizable, but not RandomAccess

Ex:-
```
public class Demo{
    psvm(){
        Set<String> ls = new LinkedHashSet<>();
        ls.add("Vijay");
        ls.add("Suresh");
        ls.add("null");
        ls.add("Vijay");
        ls.add("Kiran");
        S.O.P(ls);
    }
}
```
O/P:- [Vijay, Suresh, null, Kiran]

☆ **TreeSet** Impx

- It is implementation class of Set interface present since JDK 1.2V
- TreeSet is not an Hash based collection
- TreeSet doesnot store heterogenous value, it

To sort we need to compare element with same type of element

SURYA Gold
Date _____ Page _____

only store homogenous values (bcz of sorting)
- Tree Set is mainly used for maintaing uniquenes & sorting.
- Tree Set internally uses Comparable or Comparator interface for sorting.
- TreeSet doesnot store even a single null value,
- Tree if we try to store any null value, then it throws an exception called NullPointerException
- TreeSet maintains internal datastructure called "Balanced Tree."
- There are 4 overloaded constructor
  (i) + TreeSet()
  (ii) + TreeSet (Comparator ct) // for customized sorting
  (iii) + TreeSet (SortedSet s)
  (iv) + TreeSet (Collection c)
- It implements marker interface like Cloneable and Serializable.

```
Ex:- public class Demo {
        psvm() {
        Set<String> ts = new TreeSet<>();
        ts.add("Virat");
        ts.add("Abd");
        ts.add("Maxwell");
        ts.add(null); // NullPointerException
        ts.add(145); // ClassCastException
        ts.add("Faf");
        S.o.p(ts);
        }
}
```

O/p:- [Abd, Faf, Maxwell, Virat] in sorted order by default ascending order

For descending order:

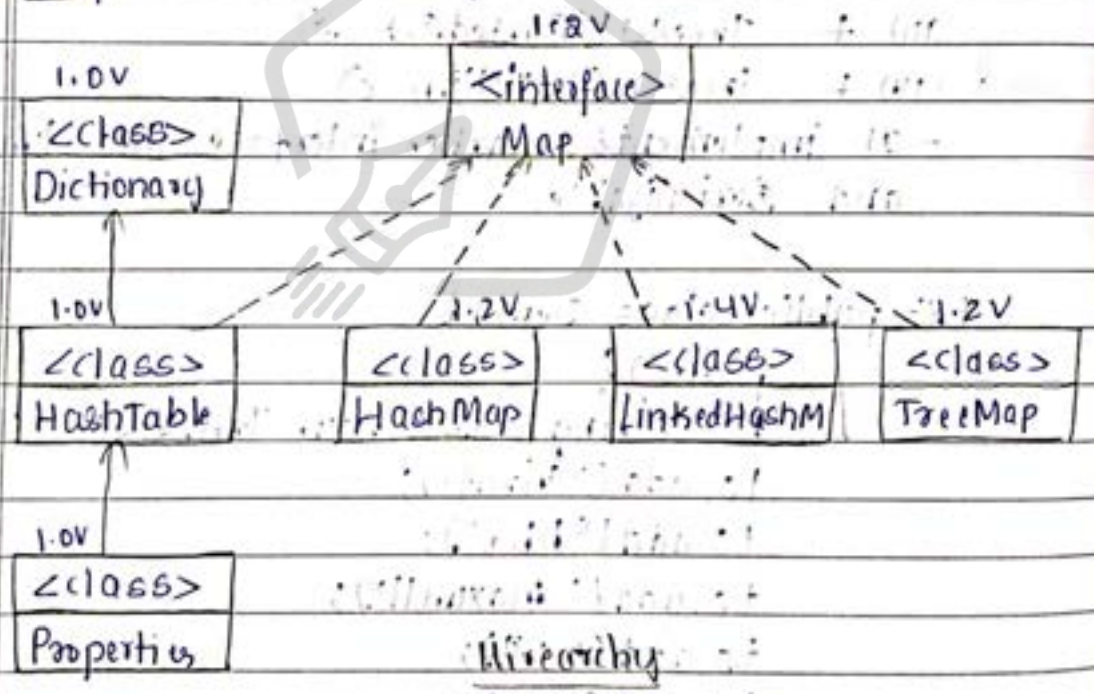
```
S.o.p(ts.descendingSet());
//[Virat, Maxwell, Faf, Abd]
```

(or)

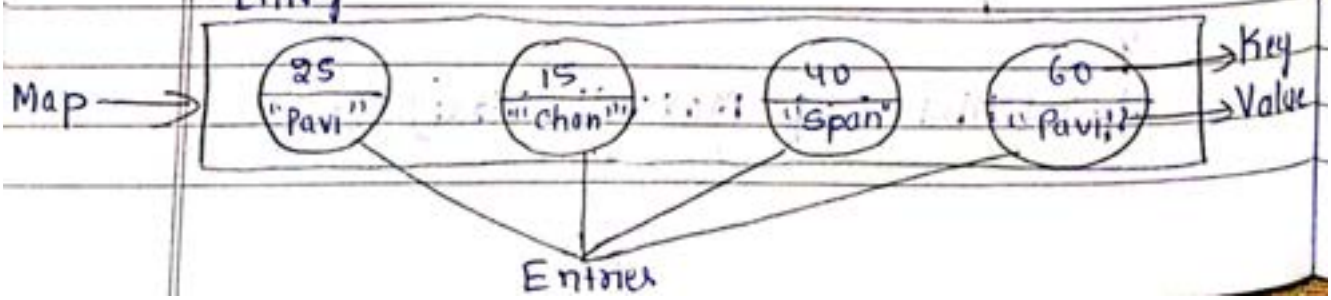
```
TreeSet<String> ts = new TreeSet<>(Comparator.reverseOrder());
S.o.p(ts);
//[Virat, Maxwell, Faf, Abd]
```

23/10/2023

★ Map



- Map is an interface present since JDK 1.2V.
- It is a collection of entries.
- Entry



- Entry is a data in the form of key and value pair, where key must be always unique.
- Entry value can be same, null but not mandatory.
- Map it is a different vertical of collection framework & it is not related to collection. Hence it doesnot shares methods of collection.

☆ Methods of Map. (Imp**)

- int size();
- boolean isEmpty();
- boolean containsKey(Object key);
- boolean containsValue(Object value);
- V get(Object key);
- V put(Key k, Value v); / V put(K key, Value v);
- V remove(Object key)
- void putAll(Map anotherMap);
- void clear();
- Set<K> keySet();
- Collection<V> values();
- Set<Map.Entry<K,V>> entrySet();

Note:- The entire Map and its functionalities are declared in java.util package.

- To create an object for Map interface implementation

```
Map<K, V> m = new HashMap();
                  HashTable();
                  LinkedHashMap();
                  TreeMap();
```

SURYA Gold

Date ____ Page ____

```
Ex:- import java.util.*;
     public class DemoMap{
       psvm(){
       Map<Integer, String> m = new HashMap<>();
       Map<Integer, String> m1 = new Hashtable<>();
       m1.put(25, "Dinga");
       m.put(40, "Gunda");
       m.put(60, "Guldu");
       m.put(25, "Dingi");
       m.put(10, "Guldu");
       S.o.p(m); // [25=Dingi, 40=Gunda, 60=Guldu,
                     10=Guldu]
       S.o.p(m.size()); // 4
       S.o.p(m.containsKey(40)); // true
       S.o.p(m.containsValue("Pavi")); false
       S.o.p(m.get(25)); // Dingi
       S.o.p(m.remove(60)); //Guldu
       m1.putAll(m);
       S.o.p(m1);
       m.clear();
       S.o.p(m.isEmpty()); //True
       }
     }
```

24/11/2023

★ Nested interface

```
Ex:- interface Map
     {
       interface Entry
       {
         K getKey();
         V getValue();
```

```
}
}
```

Map.Entry<K,V>

★ import java.util.*; or import java.util.Map.Entry;
map.Entry<K,V> Entry<K,V>

```
Map<Integer,String> m = new HashMap<>();
m.put(25, "Sukanya");
m.put(40, "Sanjay");
m.put(20, "Kunal");
m.put(10, "Sanjay");
m.put(50, "Abhishek");
Set<Integer> Key = m.keySet();
S.o.p(Key);
// key = 25 40 20 10 50

Collection<String> val = m.values();
S.o.p(val);
val = Sukanya, Sanjay, Kunal, Sanjay, Abhishek
```

★ HashMap

- It is a ~~col~~ implementation class of Map interface present since JDK 1.2V.
- HashMap is hash based map, i.e it stores object based on hashcode.
- HashMap is may or maynot follow insertion order.
- In entire map the operations like searching, sorting, insertion order and uniquiness is

maintained based on Key not on values.
- Hashmap stores only one null ~~value~~ Key.
- Initial Capacity 16 & filled ratio is 0.75%.
- It internally follows data-structure called Hashtable.
- It implements marker interface like cloneable and Serilizable.

Note:- Map cannot be iterated by using for, for-each & Iterator directly.

- There are 4 overloaded Constructor :
(i) + HashMap()
(ii) + HashMap(int capacity)
(iii) HashMap(int capacity, float loadfactor)
(iv) HashMap(Map anotherMap)

Ex:-
```
import java.util.Map;
import java.util.HashMap;
import java.util.Set;

public class HashCodeDemo{
    psvm(){
    Map<Integer,String> hm = new HashMap<>();
    hm.put(20, "Sudeep");
          (10, "Yash");
          (30, "Darshan");
          (26, "Rachitha");
          (15, "Ashika");
    // Fetching all Keys
    Set<Integer> Key = hm.KeySet();
    // iterating Keys using for each loop
```

```
for (Integer Keys : Key)
{
    String value = hm.get(Keys);
    S.o.p ( Keys + " " + value);
}
// fetching all Key & values
Set<Integer, String>> entry = hm.entrySet();
// Iterating all entries.
for ( <Map.Entry<Integer, String>> ent = entry)
{
    String value = ent.getValue();
    Integer Key = ent.getKey();
    S.o.p ( Key + " " + value);
}
```

27-11-2023

★ LinkedHashMap

- It is a implementation class of Map interface present since JDK 1.4v.
- LinkedHashMap maintains a insertion order.
- Internal datastructure maintained by LinkedHashMap is Hashtable and LinkedList.

Ex:-
```
import java.util.*;
public class Demo{
psvm(){
Map<Integer, String> lhm = new LinkedHashMap<>();
lhm.put( 1, "Vijay");
lhm.put (2, "Suresh");
lhm.put (3, "Shekhar");
S.o.p (lhm); // { 1 = Vijay , 2 = Suresh, 3 = Shekhar}
}
}
```

SURYA Gold
Date ____ Page ____

☆ TreeMap

- It is implementation class of Map interface present since JDK 1.2V
- It is used for sorting & sorting happens on based on keys not on values.
- It stores homogenious data.
- TreeMap does not store even a single null key.
- TreeMap is not an hashbased map.
- TreeMap internally uses Comparable or Comparator for sorting.

EX:-

```
psvm(){
    Map<Integer, String> tm = new TreeMap<>();
    tm.put(null, "Bijay"); // Error NPE
    tm.put(11, "Suresh");
    tm.put(33, "Shekhar");
    tm.put(12, "Kiran");
    tm.put(10, null);
    s.o.p(tm);
}// {10=null, 11=Suresh, 12=Kiran, 33=Shekar}
}
```

☆ Hashtable

- It is implementation class of Map interface present since JDK 1.0V.
- Since Hashtable is present in JDK 1.0V, we consider it as legaly class.
- Hashtable is an hashbaased map.
- It is single threaded class.
- The methods of Hashtable is synchronized.

SURYA Gold

Date _____ Page _____

- The initial capacity of Hashtable is 11 & incremental capacity is 0.75 for load factor (filled ratio).

- Key = city, Value = Customer

```
public class City {
    String cname;
    long pincode;
    public City( String name, long code) {
        cname = name;
        pincode = code;
    }
    @Override
    public String toString() {
        return cname + ", " + pincode;
    }
}
public class Customer {
    String name;
    long contact;
    public Customer (String cname, long con) {
    name = cname;
    contact = con;
    }
    @Override
    public String toString() {
        return name + ", " + contact;
        (or) return name; // to print only name
    }
}
```

```
public class Test {
    psvm(){
        City c1 = new City("Pavi", 572225);
        City c2 = new City("Tumkur", 573002);
        City c3 = new City("Hassan", 560001);

        Customer cm1 = new Customer("Pavi", 9353728[illegible]
        Customer cm2 = new Customer("Chandu", 6365[illegible]
        Customer cm3 = new Customer("Moni", 8431926090);

        Map<City,Customer> m = new HashMap<>();
        m.put(c1, cm1);
        m.put(c2, cm2);
        m.put(c3, cm3);
        S.o.p(m);
    }
}
```

28/11/2023

- To maintain the uniqueness by overriding hashcode and equals().

Note:- In String class & all wrapper class Hashcode() & equals() are already overridden. Hence, by default it maintains uniqueness when we store it in set.

```
Ex: public class Product {
        String pname;
        int pid;
        public Product(String name, int id){
        pname = name;
        pid = id;
        }
}
```

```
@Override
 public String toString(){
    return this.pid + " " + this.pname;
}

@Override
 public int hashCode(){
    return pid;
}
@Override
 public boolean equals(Object obj){
   if (this==obj)
       return true;
   if (obj==null || this.getClass() != obj.getClass())
       return false;
   Product other_product = (Product) obj;
   return this.pid == other_product.pid;
}

public class ProductCol{
   psvm(){
    Product p1 = new Product(112, "Shoes");
    Product p2 = new Product(113, "Watch");
    Product p3 = new Product(114, "Bag");
    Product p4 = new Product(113, "Watch");
    Product p5 = new Product(115, "Shoes");
    Set<Product> p = new HashSet<>();
    p.add(p1);
    p.add(p2);
    p.add(p3);
    p.add(p4);
    p.add(p5);
```

```
S.o.p (s.p);
}
}
```

o/p:- [112 Shoes, 113 Watch, 114 Bag, 115 Shoes]

☆ Sorting.

Arranging the elements in ASC Or DSC order.

Ex:-
```
public class SortDemo{
    psvm(){
        List<String> lst = new ArrayList<>();
        lst.add("Pavi");
        lst.add("Chandhu");
        lst.add("Nishu");
        S.o.p(lst); [Pavi, Chandhu, Nishu]
        // To sort in ascen order
        Collections.sort(lst);
        S.o.p(lst); // [Chandhu, Nishu, Pavi]
        // To sort in DESC order
        Collections.sort(lst, Collections.reverseOrder());
        S.o.p(lst); // [Pavi, Nishu, Chandhu]
    }
}
```

☆ Collections (I, 2V)

- It is a class present in java.util package
- It is mainly used for sorting.
- Methods

(i) + static void sort(List ls) ⟶ Comparable

(ii) + static void sort(List ls, Comparator c)

To sort SE we use Target

SURYA Gold
Date _____ Page _____

★ Comparable

- It is an interface present in default package called java.lang
- It is present since 1.2v
- Comparable is an functional interface which contains only one abstract method.
  - public int compareTo(T ele); → element Type

Ex:- • Comparable<String>
  + int compareTo(String ele)
  • Comparable<Product>
  + int compareTo(Product ele).
  • Comparable<Long>

- String class and All the wrapper Classes implements the comparable interface. Hence compareTo() is already overridden in all these classes.

9/11/2023

- The return type of compareTo() is int, which probably returns +1, -1, 0.

Ex:-
```
public class Demo{
    psvm(){
        Integer i = 40;
        Integer i1 = 80;
        S.o.p(i.compareTo(i1)); // -1 (i<i1)
        Double d = 40.31;
        Double d1 = 30.13;
        S.o.p(d.compareTo(d1)); // 1 (d > d1)
        Long l = 135368;
        Long l1 = 135368;
        S.o.p(l.compareTo(l1)); // 0. (l=l1)
```

Invoking Object — Passed object

Ex:- Overriding CompareTo()

for d & d1 values ⇒ d.compareTo(d1)

```
public int compareTo(Object obj){
    if (this > obj)
    {
        return 1;
    }
    else if (this < obj)
    {
        return -1;
    }
    else
        return 0;
}
```

Ex:- public class City implements Comparable<City>

```
    psv long pin;
        String name;
        public City(String cname, long pincode){
            name = cname;
            pin = pincode;
        }
        @Override
        public String toString(){
            return this.name+" "+this.pin;
        }
        @Override
        public int compareTo(Object o){
            if(this.pincode > o.pin){
                return 1;
            }
            else if(this.pin < o.pin){
```

SURYA Gold

Date _____ Page _____

```
    return -1;
}
else
    return 0;
/* Long l = this.pincode;
    return l.compareTo(o.pin);
*/
/* return this.name.compareTo(o.name);
    // to sort based on name, we can sort
    based on only one parameter either name/pin
*/
}
}

public class Test {
  psvm() {
    City c1 = new City("Bdore", 560078);
    City c2 = new City("Hassan", 573201);
    City c3 = new City("Tumkur", 572101);
    City c4 = new City("Turuvekere", 572225);
    List<City> lst = new ArrayList<>();
    lst.add(c1);
    lst.add(c2);
    lst.add(c3);
    lst.add(c4);
    Collections.sort(lst);
    for (City var: lst)
    {
        S.o.p(var);   //[Blore 560078
    }                  Tumkur 572101
}                      Turuvek 572225
}                      Hassan 573201
```

SURYA Gold
Date _____ Page _____

☆ Sorting Map.

- Map can be sorted only by using TreeMap

```
Ex: public class Player implements Comparable<Player>{
    String name;
    int jersey;
    public Player(String pname, int jno){
        name = pname;
        jersey = jno;
    }
    @Override
    public String toString(){
        return this.name + " " + this.jersey;
    }
    @Override
    public int compareTo(Object o){
        Integer i = this.jersey;
        return i.compareTo(o.jersey);
    }
}

public class Team {
    String team;
    String owner;
    public Team(String team, String owner){
        this.team = team;
        this.owner = owner;
    }
    @Override
    public String toString(){
        return this.team + " " + this.owner;
    }
}
```

SURYA Gold

Date ____ Page ____

```
public class Test {
    psvm() {
    Player p1 = new ("Virat", 18);
    Player p2 = new ("Dhoni", 7);
    Player p3 = new ("Rohit", 45);
    Player p4 = new ("Rahul", 1);
    Team t1 = new Team ("RCB", "A");
    Team t2 = new Team ("LSG", "B");
    Team t3 = new Team ("MI", "C");
    Team t4 = new Team ("CSK", "D");

    Map<Player, Team> m = new TreeMap<>();
    m.add(p1, t1);
    m.add(p2, t4);
    m.add(p3, t3);
    m.add(p4, t2);
    S.o.p(m); //
    Set<Map.Entry<Player, Team>> s = m.entrySet();
    for (Map.Entry<Player, Team> var : s)
    {
        Team t = var.getValue();
        Player p = var.getKey();
        S.o.p(p + " " + t);
    }
}
}
```

Runtime error — check for syntax
Runtime exception —

SURYA Gold
Date ____ Page ____

30/11/2023

☆ Exception

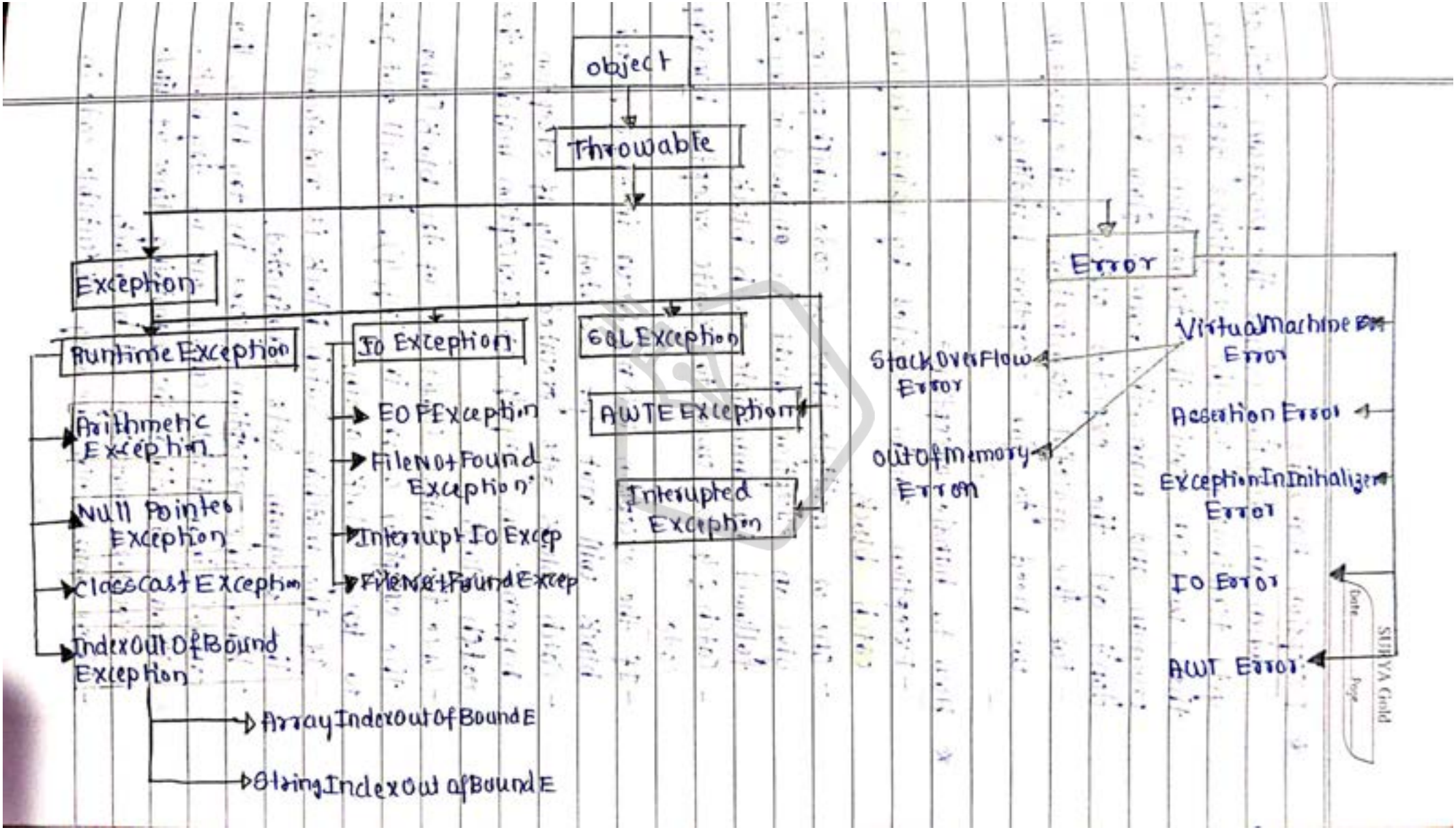
- An unwanted / un-expected event which disturbs the normal flow of program is called Exception.
- It is a runtime interuption which makes the program to stop from executing.
- Exception always occurs only during runtime when it occurs, the program gets terminated immediately.
- Exception never occurs at compile time
- Technically, exception is a class, whenever any exception occurs, JVM creates a object of respective class & throws an exception.

☆ Exception Hierarchy

object

Throwable

checked exception :- cl



Sharath Bhairaraju

SURYA Gold
Date _____ Page _____

☆ Exception Handling :

- When an exception occurs the program definitely get termination. If the exceptions are handled then, we can avoid the termination of the program and maintain the normal flow but we don't get the expected output.
- For the graceful termination of the program we need to go for exception handling.
-

☆ How to handle exception ?

- Exception can be handled by using try and catch block.
- One or multiple lines of code which possibly give an exception must be enclosed in try block followed by immediate catch block.
- Catch { } is used to write exception handling code
- If the exception did not occur in try { }, catch block will not get executed.
- Catch block gets executed only if exception happens in try { } and if exception matches the catch block.
- Once the control comes out of try block (when the exception occurs in try { }), it will not go to try block again.
- If the matching catch block is not found then the program gets terminated.
- When the exception occurs in try block, then control immediately comes out of try block, without executing the remaining lines of code & executes matching catch block.
- There must not be any executable code in

between try and catch block.

Ex: 1:- try {
1. Hi
2. Hello
3. Welcome - Exception
4. Pentagon [Never Executed]
}
Catch (Exception e) {
5. Sorry
6. Exception
}

O/p:- Hi
Hello
Sorry
Exception

Ex: 2:- try {
1. Hi
2. Hello
}
catch(Exception e) {
3. Sorry
4. Exception
}

No exception

Never Executed

O/p:- Hi
Hello

Ex: 3: try {
}
S.o.p(); //Error
catch(Exception e) {
}

Ex:- Handling Exception by using try & catch
public class ExcDemo {
psvm() {

```
int a = 50;
int b = 0;
double res = a/b; //Exception (Arithmetic Exception)
S.o.p(res);
S.o.p("Rest of the code");
}
}
```

To handle exception

```
psvm(){
    int a = 50;
    int b = 0;
    double res;
    try{
        res = a/b;
        S.o.p(res);
    }
    catch(ArithmeticException e){
        b = 10;
        res = a/b;
        S.o.p("The result is " + res);
    }
    S.o.p("Rest of the code");
}
}
```

o/p:- The result is 5.0
Rest of the code

☆ try{ } with multiple catch block{ }

```
try{
    Arithmetic Operation;
```

```
        Database Operation:
        File Operation
}
catch (ArithmeticException e)
{ ____
}
catch (SQLException e)
{ ____
}
catch (FileNotFoundException e)
{ ____
}
```

```
Ex:- public class Demo{
    psvm(){
        String s = null;
        int[] a = {10, 20, 30};
        int c = 50;
        int d = 0;
        try{
            S.O.P(a[4]);
            ~~S.O.P(charAt~~ S.O.P(s.charAt(3));
            int res = a/b;
            S.O.P(res);
        }
        catch(ArrayIndexOutOfBoundsException e){
            S.O.P("Exception is handled");
            S.O.P(a.length[1]);

        catch(NullPointerException e){
            s = "Pavithra";
            S.O.P(s.charAt(3));
        }
```

```
catch (ArithmeticException e) {
    b = 10;
    res = a/b;
    S.o.p (res);
}
S.o.p("Rest of the code");
}
}
```

O/p :- Exception handled
20
Rest of the code

Ex:- How to handle?

- Exception means, jvm creates object of appropriate exception class & that object is thrown.
- Reference type used in the catch block must belong to exception category. That is, it must be directly or indirectly a sub class of throwable.

★★★ - If there is multiple catch {} , then we have to follow the sequence **from subclass to superclass** but not from super-class to sub-class.

Subclass → Superclass ✓

Superclass → Subclass (compile time error)

- Because incase of super to subclass, whenever we get an exception in try block, all the exception can be handled by super-class & it won't allow sub-class catch block.

- Hence the subclass catch{} will become unreachable code, which is not supported in java. Hence it throws compile-time error.

Note: Whenever the handling scenario is same for multiple exception, then we can use single catch block with superclass reference.

```
try{
    Math Operation
    Array Operation
    Database Operation
    File Operation
}
```

Subclass	Superclass
catch(ArithmeticE e)	catch(Exception e){
{ --	--
}	}
catch(ArrayIOBEX e)	catch(ArrayIOBE e)
{ --	{
}	}
catch(Exception e)	catch(ArithmeticE e)
{ --	{
}	}
Superclass	Subclass, Hire compiletime error

02/12/2023

- From JDK 1.7 version onwards we can handle multiple exception within the same catch block.

Ex:- try
{
math operation

```
    DB Operation
    File Operation
}
catch (ArithmeticException | Exception e)
{
    Handling code
}
```

★ Nested try and catch

```
try{
    Math operation
    try{
        DB operation
        try{
            File Operation
        }
        catch(FilenotFoundException e){
            // Handling code
        }
    }
    catch(SQLException e){
        // Handling Code
    }
}
catch(ArithmeticException e){
    // Handling Code
}
```

1st 2nd 3rd

- We can write try & catch{} inside another try{}, catch{}, finally{}.
- If the handling scenario is same for multiple

exceptions, then we write single catch block.
- If the handling scenario is different for multiple exceptions, the we write multiple catch block.

* finally block
- It is used in exception handling.
- It is always gets executed irrespective of whether ~~ns~~ the exception ocair or not.
- Single try{} can have Only one finally{}.
- There can have try{} without catch{}, in this case finally{} must associate with the try{}
- Any mandatory code which has to be executed irrespective of exception must be writtened inside finally{}

Note:- All the costly resource [heavy weight object] are usually closed inside finally block.

Ex:- try {
Validate Card operation
Validate account operation
Validate balance
Withdraw operation
Process amount
}
catch (InsuffientFund e) {

}
catch (ATMException e) {

}

finally {
return Atm Card Operation
}

04/12/2023

☆ Costly resource | Heavyweight Object
- The resource which makes use of system properties in the form of stream is costly resource.
Ex: Scanner → Keyboard
FileReader → Existing file
FileWriter
PrintWriter
DB Connection interface.
- It is a good practice to close the costly resource otherwise the performance of an application will be reduced.
- In general all the costly resource are closed with the help of finally block.
- If the costly resource are not closed then it won't throw any exception/error, but the performance will become slow (Warning:- Resource Leak)

```
EX:- FileReader f = new FileReader("data.txt");
     f.read();
     finally {
         if(f != null) {
             f.close();
         }
     }
```

- read() :- It is used to read the characters from the given file.
• The return type of read() is int, which returns the ASCII values of the characters.

• read() returns -1, when it comes to end of the file.

```
Ex:- import java.io.*;
    public class CostResDemo{
        psvm() throws IOException {
            FileReader fr = null;
            try{
                fr = new FileReader("D:\\Java.text");
                int i = fr;
                while((i = fr.read()) != -1) //To iterate all characters
                {
                    S.o.p((char)i);   fr.read() = Firstchargit
                }
            }
            catch(IOException e){
                e.printStackTrace();
            }
            finally{
                if(fr != null){
                    try{
                        fr.close();
                    }
                    catch(IOException e){
                        e.printStackTrace();
                    }
                }
                S.o.p("Costly Resource File Reader closed");
            }
        }
    }
```

SURYA Gold
Date _____ Page _____

* Types of Exception:
  (I) Checked Exception
  (II) Unchecked Exception

(I) Checked Exception
- Those exception for which the compiler checks at the compile time, whether the exception is handled by the developer inside the code or not.
- If the exception is not handled, then the compiler gives an error & force the developer to handle the exception.
- The exception certainly happens only at a runtime but the compiler checks whether it is handled or not.
- Checked exception classes never extends Runtime exception class.

Ex:- IOException, AWTException, SQLException, InteruptedException.

(II) Unchecked Exception
- Those exception for which the compiler doesnot checks at the compile time, whether the exception is handled by the developer inside the code or not.
- If the exception is not handled, then the compiler doesnot given any error & will not force the developer to handle the exception.
- Unchecked exception classes always extends Runtime exception class.

Ex:- RuntimeException class sub classes.
i.e ArithmeticException
ClassCastException etc

★ throws and throw

- **throw :-**
  It is a keyword which is used to throw an exception explicitly in the program inside a method (or) block object

- **throws :-**
  It is a keyword used in the method signature to indicate the exception.

- throw is used to create & return one single exception object at a time
- throws can be used to indicate multiple exception at a time

throwe ⇒ Ex:-

```
public class ExceptionDemo{
    static void div(int n) throws ArithmeticException
    {
        int i=100;
        double div = i/n;
        S.o.p(div);
    }
    psvm() {
    try{
        div(0);
    }
    catch(ArithmeticException e){
        e.printStackTrace();
    }
}
}
```

06/12/2023

- Ex:- Using throw.

```
public class throwDemo{
public void checkSquare(int n){
  if (n<0)
  {
    throw new ArithmeticException("zero");
  }
  else
  {
    S.O.P("The Square of num is " + n*n);
  }
}
}
public class Test{
psvm(){
  try{ throwDemo t = new throwDemo();
    try{
      t.checkSquare(10);
      t.checkSquare(-1);
    }
    catch(ArithmeticException e){
      S.O.P(e);
    }
  }
}
```

Note:- Sometimes, when we have to throw Exception object manually, then we need to use throw Keyword

- Ex for throws:

```
class Demo{
    psvm(){
        thread.sleep(500);
    }
}
```

Two ways handle

```
psvm(){
  try{
      thread.sleep(500);
  }
  catch(InteruptedException e){
    S.O.P("message:"+e);
  }
}
```

```
psvm() throws
InteruptedException{
    thread.Sleep(500);
}
```

- In the above example we are handling the exception in two different ways.
  (i) By using try{} and catch{}.
  (ii) By using throws Keyword.

Note:- We can use throws to handle the but it is not good practice because of 2 reasons.
(i) In-case of throws we can't able to write handling code.
(ii) In case of throws we are delegating the respons-bility of handling exception to the method caller, which is not a good practice.

☆ Custom Exception / User-defined Exception Impx

- To create Unchecked Exception

```
public class PentagonSpace {
    + void register(int age):{
        if(age>50||age<20){
            throw new InvalidAgeException("Your age
                                is below/Above 50");
        }
        else:
            S.O.p("Welcome to Pentagon Space");
    }
}
class InvalidAgeException extends RuntimeException{
    InvalidAgeException (String msg)
    {
        Super(msg);
    }
}

public class Test {
    psvm(){
        PentagonSpace p=new PentagonSpace();
        p.register(25); //welcome to Pentagon Space
        p.register(19);
        p.register(54);  } InvalidAgeException occurs
    }
}
```

- In the above example it is not mandatory to handle the exception. Since InvalidAgeException is

unchecked E.
- when we don't handle the Exception then JVM will handle the exception by using default Exception handler.

Write a difference b/w final, finalize, finally.

	final	finalize	finally
☆	final is keyword used to declare & apply restriction on a class, method, variable	☆ finalize is a method used to indicate the object that is de-referred	☆ finally is a block used in exception handling to execute the important code irrespective of exception.
☆	It is used with class, methods and variables	☆ finalize() is used with the objects	☆ It is related to try & catch e.g. in exception handling
☆	final members (variable, methods) can be executed while we call it final class cannot be extended. final methods & variables cannot be modified	☆ It is executed just before the object is destroyed.	☆ It is executed as soon as the try-catch is executed. Its execution is not depende on exception.

06/12/2023

☆ Multi-tasking

Performing a several task simultaneously is called multi-tasking.

There are two types of multitasking

(i) Process based multi-tasking. (PDM)

(ii) Thread based multi-tasking.

(i) Process based multi-tasking

- Executing several task simultaneously where each task is a separate independent process.
- It is best suitable for OS level.

Ex:- Listen to music & Browse the internet at the same time.

(ii) Thread Based multi-tasking

- Executing several task simultaneously where each task is separate independent part of same program.
- It is best suitable for programmatic level.

Ex:- While browsing we can navigate through the web pages & at the same time we can download a file.

☆ Multi-threading (Imp**)

It is a process of executing multiple threads simultaneously.

Multithreading is mostly used in gaming applications and animations etc.

Thread:-

- A thread is a lightweight ~~pr~~ sub-process, a smallest unit of processing.

- Threads are independent.
- If the exception occurs in one thread, it doesn't affect other thread.
- Thread is a separate path of execution.
- Each Threads execute in separate runtime stack memory.

★ Multi-processing
- Each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication b/w process is high
- Switching from one process to another require some time for saving, loading registers, memory maps, updating lists etc.

The multi-threading and multi-processing are used to achieve multi-tasking.

★ Thread Constructors

```
public Thread()
public Thread(String name)
public Thread(Runnable r)
public Thread(Runnable r, String name)
```

Note:- Thread is a class present in java.lang package.

★★ There are 2 different ways to create a thread
(i) By extending to Thread class.
i.e class Demo extends Thread.
(ii) By implementing Runnable interface.
i.e class Demo implements Runnable.

SURYA Gold

Date _____ Page _____

(1) By extending thread class

Ex:-
```
public class MyThread extends Thread{
    @Override
    public void run(){
        S.o.p("Thread is running");
    }
psvm(){
    MyThread t = new MyThread();
    t.start(); //Thread object is created
}
```

(1) By extending thread class

```
Ex:- public class Mythread extends Thread {
        @Override
        public void run() {
            S.o.p("Thread is running");
        }
    psvm() {
        MyThread t = new MyThread();
        t.start(); //thread object is created
    }
}
```

job of thread

08/12/2023

```
Ex:- public class Mythread extends Thread {
    @Override
    public void run() {
        for(i=0; i<10; i++) {
            S.o.p.n("child Thread is running");
        }
    }
    psvm() {
        Mythread t = new Mythread();
        t.start();
        for(int i=0; i<10; i++) {
            S.o.p.ln("Main thread is running");
        }
    }
```

O/p:- Main thread is running
Main thread is running
Main thread is running

child Thread is running

☆ Thread Scheduler :-

- It is a part of JVM.
- It is responsible to schedule threads i.e. if multiple threads are waiting to get the chance of execution then in which order threads will be executed is decided by thread scheduler.
- We can't expect exact algorithm followed by thread scheduler it is varied from JVM to JVM hence we can't expect threads execution order and exact output.
- Hence, whenever the situation comes to multithreading there is no guarantee for exact output be we can provide several possible outputs.

☆ Methods of thread

(I) void start() :- It is used to start the execution of thread.

(II) void run() :- It is used to do an action for a thread.

(III) static void sleep() :- It sleeps a thread for a specified amount of time.

(IV) void join() :- It waits for a thread to die join

(v) static void yield() :- It causes the currently executing thread object to pause & allow other object to execute temporarily.

Ex:-
```
class MyThread extends Thread{
    void run()
    {
        S.o.p("Child Thread is running");
    }
}
class Test{
psvm(){
    MyThread t = new MyThread();
    t.run(); // Thread object is not created
    S.o.p("Main thread is running");
}
}
```

- In the above code we are replacing t.start() with t.run().
- In this case we will get the o/p [child thread is running & main thread running], here JVM won't create two thread objects instead the entire execution taken by main thread (main method).
- run() will be executed as normal method

Note:- To define a thread, it is mandatory to invoke thread class start() which internally calls run().
- We can overload run method, in this case when

we invoke start method, internaly start() invoke zero parameter run().

```
Ex:- class Mythread extends Thread {
        @void run()
        {
          S.o.p("Child thread");
        }
        @Override
        void start() {
          super.start(); //child thread created
          S.o.p("Child start method");
        }
    }
    class Test {
    psvm() {
        MyThread m = new Mythread();
        m.start();
        S.o.p("main Thread");
    }
    }
```

- Can we override start()?

Yes, we can override start(). But, in this case we are not allowing Thread class start() to execute. Hence thread object will not be created & start() executes like normal method.

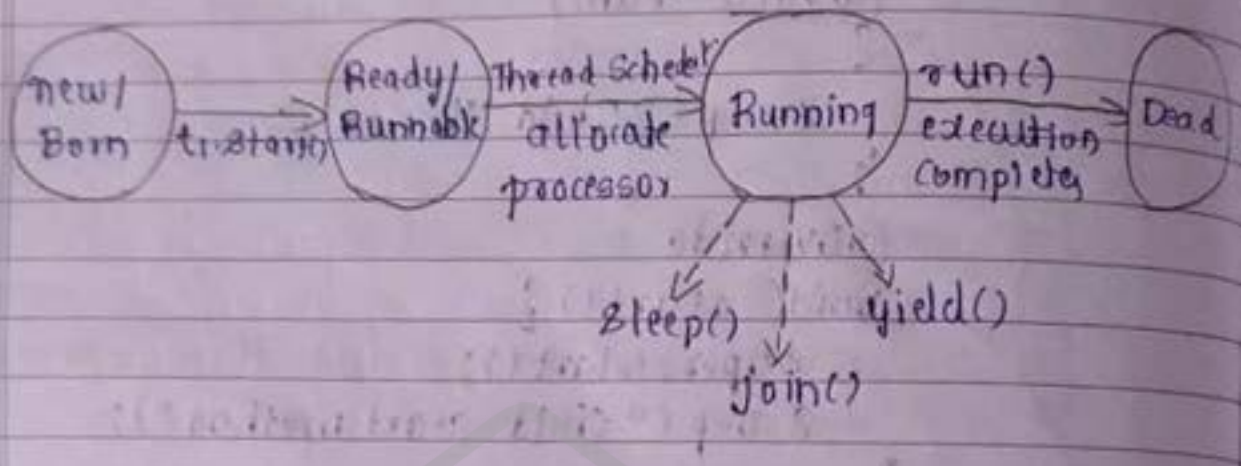
e.printStackTrace() :- Exception Name, description, location of file
e - Exception name, description
e.getMessage() - Exception name

SURYA Gold
Date _____ Page _____

10/12/2023

☆ Thread Life Cycle (Imp★★★)

MyThread t1 = new MyThread();



(I) New :-

Whenever a new thread is created, it is always in the new state & allocated with necessary resource.

For a thread in the new state, the code has not been run yet and thread has not begun its execution.

(II) Ready/Runnable

- Once the thread class start() is invoked it enters into runnable state.
- In this state the thread is ready to run, which is moved to runnable state.
- In the runnable state, the thread may be running / may be ready to run at any given instant of time.
- It is the duty of the Thread Scheduler to allocate the processor however it is not necessary to executing at this moment due to the compition

with other threads for CPU time.

(iii) Running:-
- when the Thread Scheduler allocates processor the thread gets the CPU, it moves from runnable to running state.
- Generally, in this phase thread enters into running & actively executing the code & performing its task.

(iv) Dead (Terminated)
- A thread reaches the termination state because of the following reasons:-
• When a thread has finished its job, then it exits / terminate normally.
• Abnormal termination :- It occurs when some unusual events such as an unhandled exception / segmentation fault.

Note:- A terminated thread means a thread is no more in the system. In other words, a thread is dead & there is no way one can re-born the dead thread.
- Whenever we try to restart the thread it throws an exception called IllegalThreadStateExc-eption.

Ex:-
```
MyThread t1 = new MyThread();
t1.start();
t1.start(); //IllegalThreadStateException
```

(II) Creating thread Object by implementing Runnable Interface.

```
class MyThread implements Runnable {
    @Override
    + void run() {
        S.o.p("Child Thread");
    }
    psvm() {
    MyThread m = new MyThread();
    Thread t = new Thread(m);
    t.start();
    }
}
```

Runnable:- It is a functional interface which is present in java.lang package.
- Runnable interface contains only one abstract method called run().

Note:-
(I) By extending Thread class

```
class MyThread extends Thread {
}
```

In-case of this, we are missing the benefits of inheritance with respect to class (only inherit thread class)

(II) By implementing Runnable
class MyThread extends another class implements Runnable {
}

In-case of this, we can utilise of the benefits of Inheritance with respect to class.

1/2/2023

* To set and get the name of the thread
  We can set and get the name of the thread by using getName & setName().

Ex:- psvm() {
    System.out.println(CurrentThread().getName());//main
    CurrenThread().setName("guldu");
    MyThread m = new MyThread();
    m.start();
    S.o.p(CurrentThread().getName());//guldu
}

* Thread Priorities (Imp**)
  - It always starts from 1 to 10

3 types of priorities
(i) MIN PRIORITY - 1
(ii) NORM PRIORITY - 5
(iii) MAX PRIORITY - 10

- In java, all the threads have a priorities.
- In most of the cases Thread Scheduler allocates the processor based on priority.
- JVM assign the priorities for each thread.
- The default priority assigned by JVM is "5"

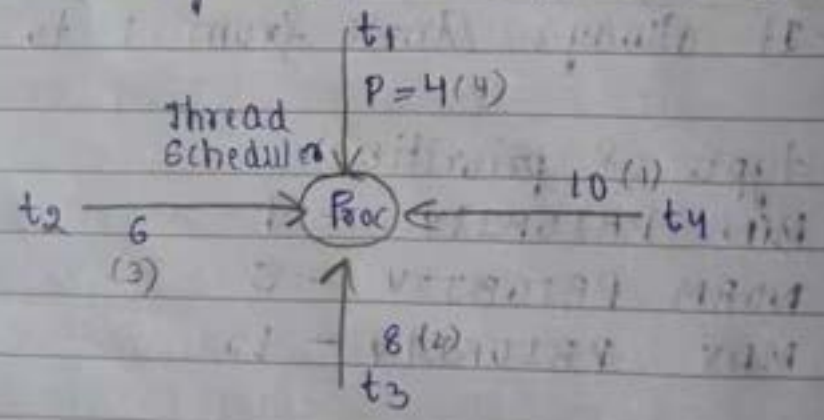
- All the child threads inherits the default priorities from parent threads.
- In java not only JVM, a java programmer can also assign priorities explicitly by using setPriority().

```
+ final void setPriority(int PriorityNumber)
{
}
```

- we can get the priority by using getPriority()

```
+ final int getPriority()
{
    return priority;
}
```

- The maximum (highest) priority is 10 & lowest priority is 1.
- When the multiple threads having same priority then ThreadScheduler will decide which thread has to be executed first based on internal algorithm (FCFS, Round-Robin).



Ex:- public class MyThread extends Thread {
    @Override
    public void run() {
        S.o.p("Child thread");
    }

Here if set priority for main thread it will be inherited to child thread also
i.e cur

```
psvm(){
    MyThread m = new MyThread();
    m.setPriority(10); //Setting Priority to child thread
    m.start();
    S.o.p("main thread");
}
}
```

o/p:- Child Thread
Main Thread.

Note:- In case of multi-threading we can't predict the exact o/p. Since JVM changes from System to System.

```
- psvm(){
    CurrentThread().setPriority(8);
    MyThread m = new MyThread();
    m.start();
    S.o.p(m.getPriority()); // 8
```

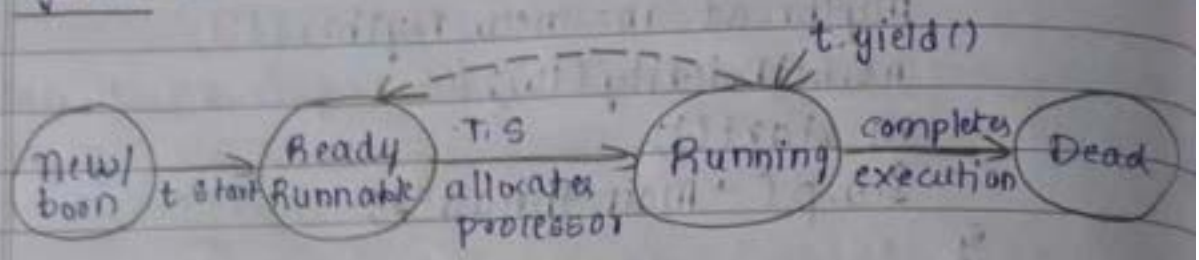
In the above program, we are setting priority for main thread which will be inherited to all child threads. Hence thread scheduler will decide which thread has to be executed

Methods to prevent the execution of thread

- There are 3 important methods:-

(i) yield():
(ii) join()
(iii) sleep()

(1) yield() :-



- yield() is used to pause the current execution of a thread to give the chance for waiting thread of a same priority, if there is no waiting thread / all the waiting threads have low priority, then the same thread will continue its execution.
- If the waiting threads having same priority, then the TS allocates the processor based on the algorithm.

Ex:-
```
public class MyThread extends Thread {
    @Overide
    public void run() {
        for(int i=1; i<=10; i++) {
            S.O.P("Child thread");
            Thread.yield();
        }
    }

    psvm() {
        MyThread m = new MyThread();
        m.start();
        for(int i=1; i<=10; i++) {
            S.O.P("Main");
        }
    }
}
```

O/p:- child thread.
10 times main
9 times child thread.

12-12-2023

(ii) join()

- If a thread t_1 needs to wait for another thread to complete the execution, then we need to use join().
- Once we invoke join method on thread t_1 then t_1 thread enters into waiting state (blocking for join). And it will be in waiting state until the thread t_2 completes the execution.
- There are ~~or~~ 3 overloaded varients of join().

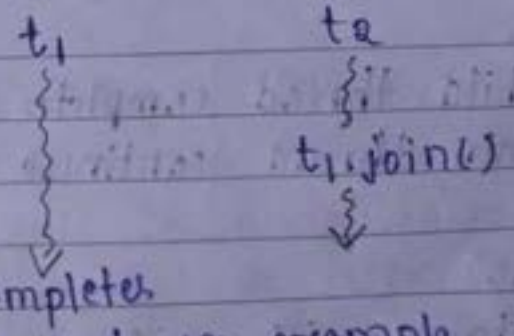
(i) + final synchronized void join() throws InterruptedException { }

(ii) + final synchronised void join(long ms) throws InterruptedException.

(iii) + final synchronised void join(int ns) throws IE

Note:- whenever we call any of this method it throws an Exception InterruptedException.

Ex:-



- In the above example t_2 thread will call join() on t_1 thread. Hence, t_2 thread will enter into waiting state and it will wait

unless untill the thread t_1 completes the execution

Ex:-
```
public class child extends Thread {
    @Override
    public void run() {
        for(int i=1; i<=5; i++) {
            S.o.p ("Child Thread");
        }
    }
    psvm() {
        Child c = new child();
        c.start();
        c.join(); //Here main thread waiting for c
        for(int i=1; i<=5; i++) {
            S.o.p ("Main Thread");
        }
    }
}
```

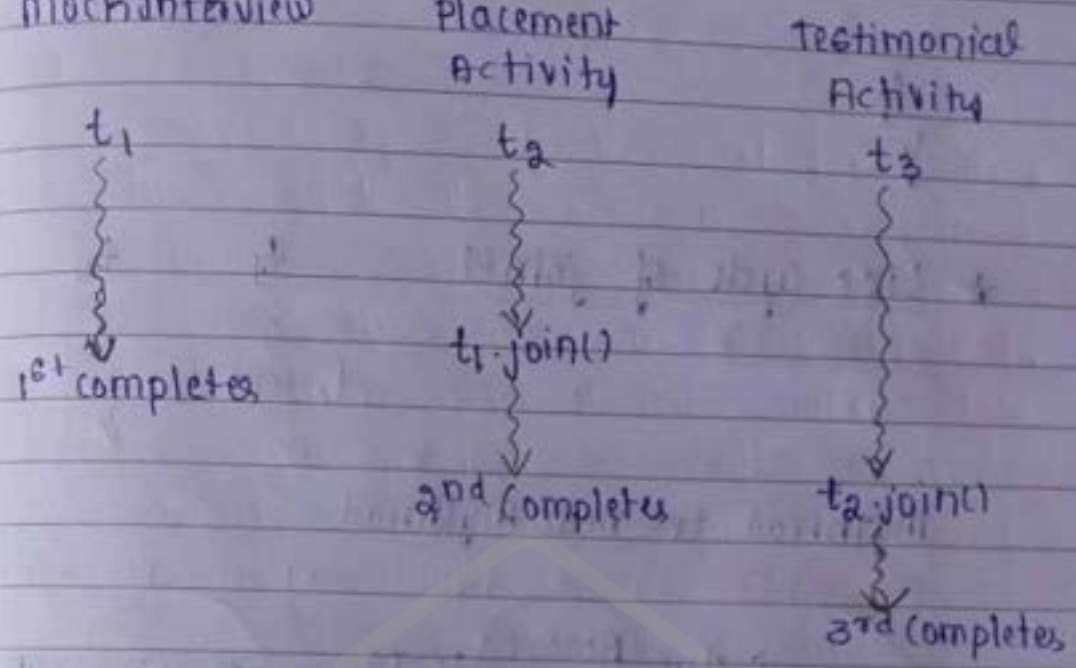
o/p :- Child Thread 5 times
Main Thread 5 times.

- Here main thread invoking join() on child thread. So, main thread will enter into the waiting state untill the child thread completes its execution.
- Once the child thread completes the execution then the main thread continue its execution.

- c.join(20000)
Here main thread will wait for 20 sec & allows child thread to complete the execution. If the child thread doesnot complete the

execution within the given time, main thread will continue its execution.

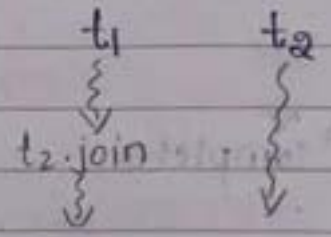
Ex:- MockInterview



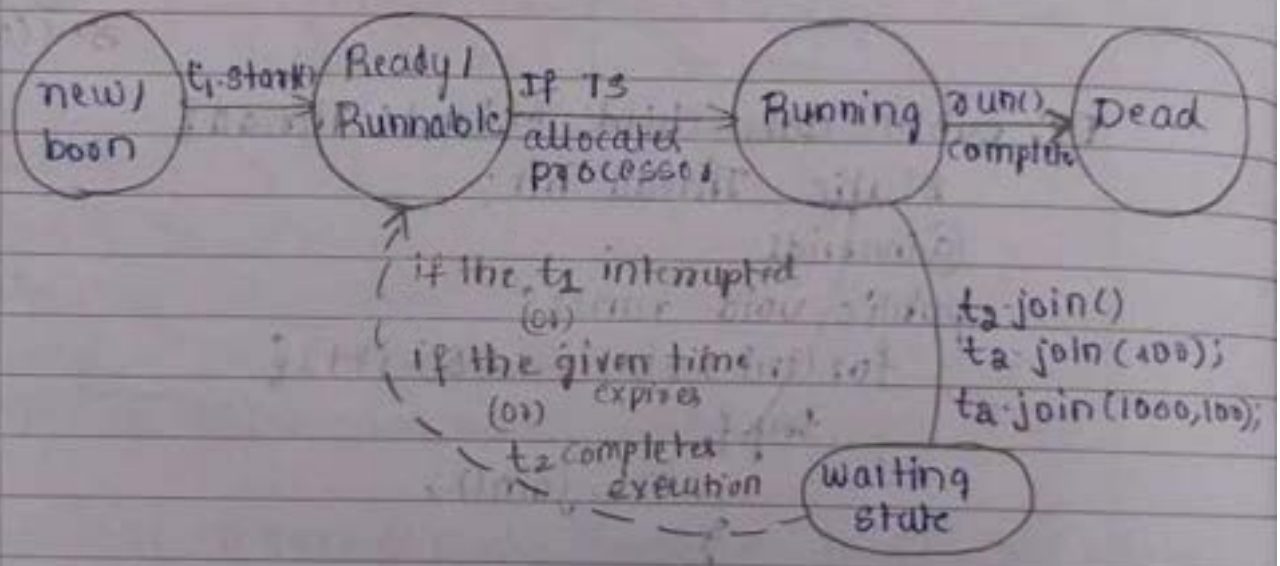

```
Ex:- public class child extends Thread{
    static Thread mt;
    @Override
    public void run(){
        for(int i=0; i<=5; i++){
            try{
                mt.join();
            }
            catch(Interupted_Exception e){
                e.printStackTrace();
            }
            Sop.p("Child thread Running");
    }
    psvm() throws InterruptedException {
        child c=new child();
        mt = CurrentThread();
        c.start();
        for(int i=0; i<5; i++)
```

```
{
    S.o.p ("main thread Running");
    Thread.sleep(1000);
}
}
}
```

★ **Life cycle of join()**



MyThread t1 = new MyThread



★ **Deadlock**

The 2 threads wait for each other where the waiting will never ends causes deadlock.

```
Ex:- class Child extends Thread {
        @Override → static Thread mt;
        public void run() {
            try {
```

```
    mt.join(); // waiting for main thread
  } catch (InterruptedException e) {
        e.printStackTrace();
  }
  S.o.p("Child thread");
}
psvm() {
    mt = CurrentThread();
    Child c = new Child();
    c.start();
    c.join(); // waiting for child thread to execute
    S.o.p("Main Thread");
  }
}
```

18/12/2023

(iii) sleep()

- This method is used to pause the current executing thread for particular amount of time.
- there are 2 overloaded varients.
  - + static native void sleep(long ms) throws IE { }
  - + static void sleep(long ms, int ns) throws IE.
- whenever we use any of this methods, it throws an InterruptedException

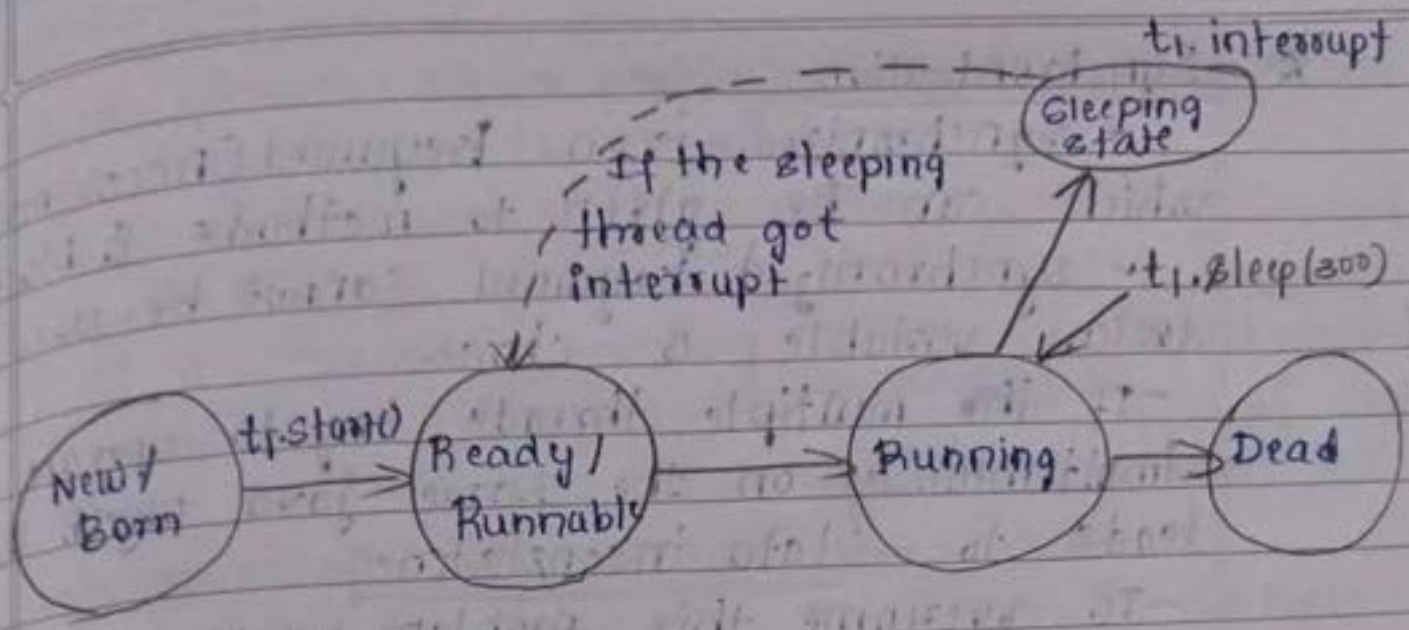
Ex:- Class MyThread extends Thread {
```
    psvm() {
        for(i=1; i<=10; i++) {
            S.o.p("slide-" + i);
            try {
```

```
        Thread.sleep(3000);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
  }
}
```

- interrupt()
  - It is used to interrupt the thread which is sleeping.

Ex:
```
public class Child extends Thread {
    @Override
    public void run() {
        for(int i=0; i<=30; i++) {
            S.o.p("Child thread");
        }
        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
            ~~e.printStackTrace();~~
            S.o.p("I got Interrupt");
        }
    }
    ~~mai~~ psvm() {
        Child c = new Child();
        c.start();
        c.interrupt();
    }
}
```

O/p:- Child thread 30 times
I got Interrupt



* Difference

	Yield()	join()	sleep()
It is overloaded?	NO	Yes	Yes
Is it final?	NO	Yes	NO
Is it throws IE?	NO	Yes	Yes
Is it native?	Yes	NO	Sleep(long ms) - native Sleep(long ms, int ns) - non native

14/12/2023

☆ Synchronisation

- synchronized is a keyword (Access modifier) which can be given to methods & block.
- synchronized keyword cannot be used to declare variable & class.
- If the multiple threads try to operate simultaneously on the same java object then it leads to data inconsistency.
- To overcome this problem we need to go for synchronized keyword.
- If the method/block is declared as synchronized then only '1' thread is allowed to execute that method/block on a given object.
- Hence, the data inconsistency problem is resolved.
- Advantage
  It is used to resolve data-inconsistency problem.
- Drawback
  - It increases waiting time of thread.
  - It decreases the performance of application.

Note :- If there is no specific requirement then it is not recommended to go for synchronised method/block.

☆☆ - Synchronization internally uses "lock" concept

- In java each and every object contains unique lock.
- Each & every object comes with one lock.
- Whenever we are executing synchronized

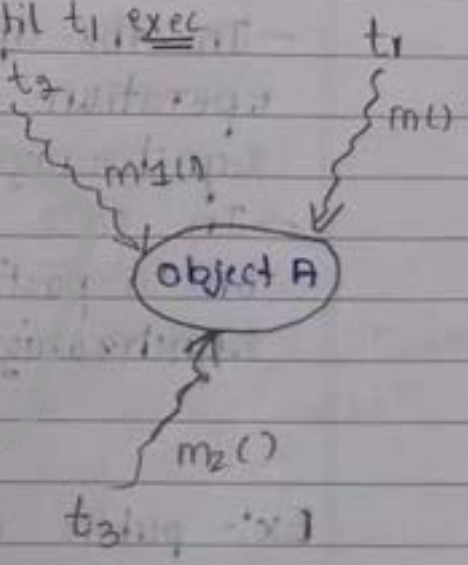
JVM provides lock for each object.

method/block on a particular object, then the lock of that object is required.

- whenever we are executing non-synchronized method/block then the lock is not required.
- If a thread is trying to execute synchronized method/block, first it occupies the lock of that object. Once the execution completes then automatically it releases the lock.
- Acquiring & releasing the lock is taken care by JVM.

Ex:- class A

```
{
  synch void m()
  {
  }
  synch void m1()
  {
  }
  void m2()
  {
  }
}
```

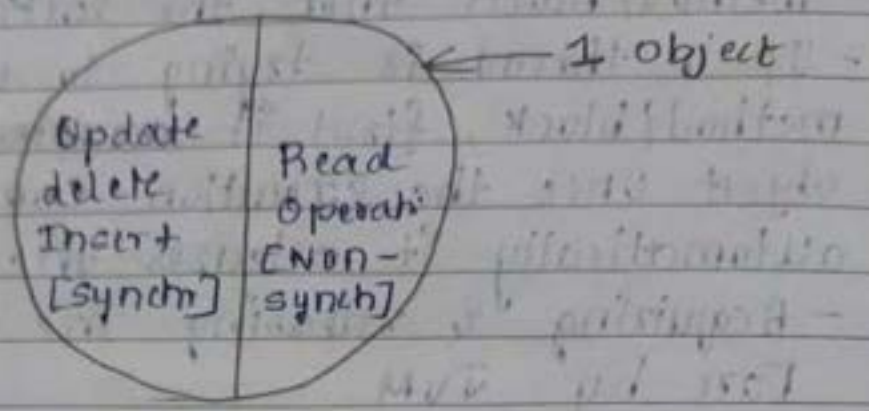


While a thread executing synchronized method on a given object then the remaining thread are not allowed to execute any synchronized methods/blocks simultaneously on same object.

But remaining threads can execute non-synchronized methods/blocks simultaneously.

Ex:- class RedBus{
    void checkAvailability () {
}

```
synchronized void reserveSeat(){
    - - - -
    - - - -
}
}
```



API

- In the application if there is any read operation then it is not preferred to use synchronized method/block.
- In case of Update, Delete, Insert operation on a particular then we need to go for synchronized method/block.

Ex:-
```
public class Display{
    synchronised void invite(String name){
    for(i=1; i<=2; i++){
        S.o.p("Welcome to:");
        try{
            Thread.sleep(500);
        }
        catch(InteruptedException e){
            e.printStackTrace();
        }
        S.o.pln(name);
    }
}
```

```
public class MyThread extends Thread {
    String name;
    Display d;
    public MyThread (Display d, String name) {
        this.d = d;
        this.name = name;
    }
    @Override
    public void run() {
        d.invite(name);
    }
}

public class MainThread {
    psvm() {
        Display d = new Display();
        MyThread m1 = new MyThread(d, "Tumkur");
        MyThread m2 = new MyThread(d, "Gubbi");
        m1.start();
        m2.start();
    }
}
```

O/p:- welcome to: Tumkur
welcome to: Tumkur
welcome to: Gubbi NO mixed O/P
welcome to: Gubbi

Difference b/w Compiler & Interpreter

15/12/2023

★ Deadlock Synchronization

Two threads wait for each other, where waiting never ends

Ex:-

```
class A {
    synchronized void d1(B b) {
        S.O.p("Thread1 get the lock of A & start executing d1()");
    S.O.p("Thread1 waits for B's lock to execute B's m1()");
    b.m1();
    }

    synchronized void m1() {
        S.O.p("Thread2 will get the lock of A & execute A's m1()");
    }
}

class B {
    synchronized void d2(A a) {
        S.O.p("Thread2 get the lock of B & starts executing d2()");
    S.O.p("Thread2 waiting for A's lock to execute A's m1()");
    a.m1();
    }
    synchronized void m1() {
        S.O.p("Thread1 will get the lock of B & execute B's m1()");
    }
}
```

@override provides extra compiler to check that method signature is matching / not.

```
public class Deadlock extends Thread {
    A a = new A();
    B b = new B();

    psvm() {
        Deadlock d = new Deadlock();
        d.meth();
    }

    void meth() {
        this.start();
        a.d1(b);
    }

    @override
    public void run() {
        S.o.P("Child Thread");
        b.d2(A);
    }
}
```

Output: Thread 1 get the lock of A & start executing $d_1()$.
Thread 1 waits for B's lock to execute B's $m_1()$.
Thread 2 get the B's lock & execute $d_2()$
Thread 2 waits for B's lock to execute A's $m_1()$.

Here, it enters to deadlock state, where Thread-1; Thread-2 are waiting for each others lock.

SURYA Gold
Date ____ Page ____

★ **Starvation**

A long waiting of a thread where the waiting ends at certain point

★ **Race Condition**

Multiple threads trying to access on a same object which leads to data inconsistency is called race condition.

★ **Inter-Thread Communication**

- Two threads communicates each other by using wait(), notify(), notifyAll() is called inter-thread communication.
- The thread which is required updation then it should call wait()
- The thread which is performing updation of object it is responsible to give notification by calling notify()
- After getting the notification the waiting thread will get those updation & continue its execution.

★★ - wait(), notify(), notifyAll() methods are present in object class but not in thread class because the thread can call these method on any common object

- public final native ~~notify~~ void notify();
- public final native void notifyAll();
- public final void wait() throws InterruptedException
- public final native void wait(long timeout)

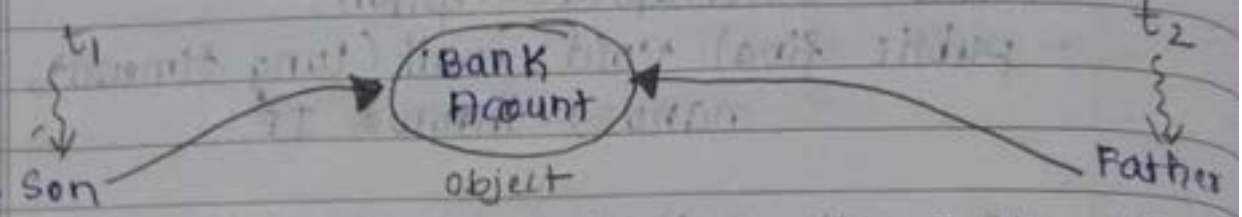
throws InterruptedException
- public final void wait (long timeout, int nanos) throws IE.

- To call wait(), notify(), notifyAll() compulsorily current thread must be owner of that object i.e current thread should have lock of that object.
- Current Thread should be in synchronized area. then only we can call wait(), notify(), notifyAll(). otherwise, we get a runtime Exception called IllegalMonitorStateException

- Once the thread calls wait() on given object first it releases the lock of that object immediately & enters into waiting state.

- Once the thread calls notify() & notifyAll() then it releases the lock of the object but not immediately.

- Once the thread call notify(), notifyAll(), wait() on any object then it releases the lock of that particular object but not all the locks it has.




```
synch withdraw(amt)                synch deposit(amt)
{                                  {
  amt < bal                          bal = bal+amt;
    acct.wait();                     acct.notify();
  bal = bal-amt;                   }
}
```

waiting state

Ex:

```
public class Customer {
    double balance = 10000;
    synchronized void withdraw (int amt)
    {
        if (amt <= balance)
        {
            try {
                S.o.p("Insufficient balance");
                wait();
            }
            catch(InterruptedException e){
                e.printStackTrace();
            }
        }
        else
        {
            balance = balance - amt;
            S.o.p("withdraw completed");
        }
    }
```

SURYA Gold

Date _____ Page _____

```
synchronized void deposit(int amt) {
    balance = balance + amt;
    S.o.p ("Amount deposited :" + amt +
            "In New balance : " + balance);
    notify();
}
}
```