Green University of Bangladesh
Department of Computer Science and Engineering (CSE)
Faculty of Sciences and Engineering
Semester: (Fall, Year:2024), B.Sc. in CSE (Day)

KSA - 1

Course Title: Object Oriented Programing
Course Code: CSE 201 Section: D1

Student Details

Name	ID
Taslim Ahmed Tamim	232002105

Submission Date : 29/12/2024
Course Teacher's Name : Dr. Muhammad Aminur Rahaman

Report Status

Marks: …… **Signature:.....**
Comments:..... **Date:.....**

What is Java?

→ An object Oriented Programming Language developed at Sun Microsystem in 1995.

Java is close to C, C++, C# which makes it easier for programmer to Java or vice versa. Java was conceived by James Grosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystem.

→ Java is a set of standardized Class libraries (Packages), that support:

* Creating graphical user interfaces (gui)
* Communicating over networks,
* Controlling multimedia data.

features of Java:

1. Platform independent:

Java has virtual machine which take the Java byte code after compilation then make it usable for any system.

2. Object Oriented:

Java is Object Oriented, because we have to make object to use any class in Java.

3. Robust:

It employs powerful memory management. Java autometically handles the memory management system. There is an absence of pointers that bypasses security dilemmas.

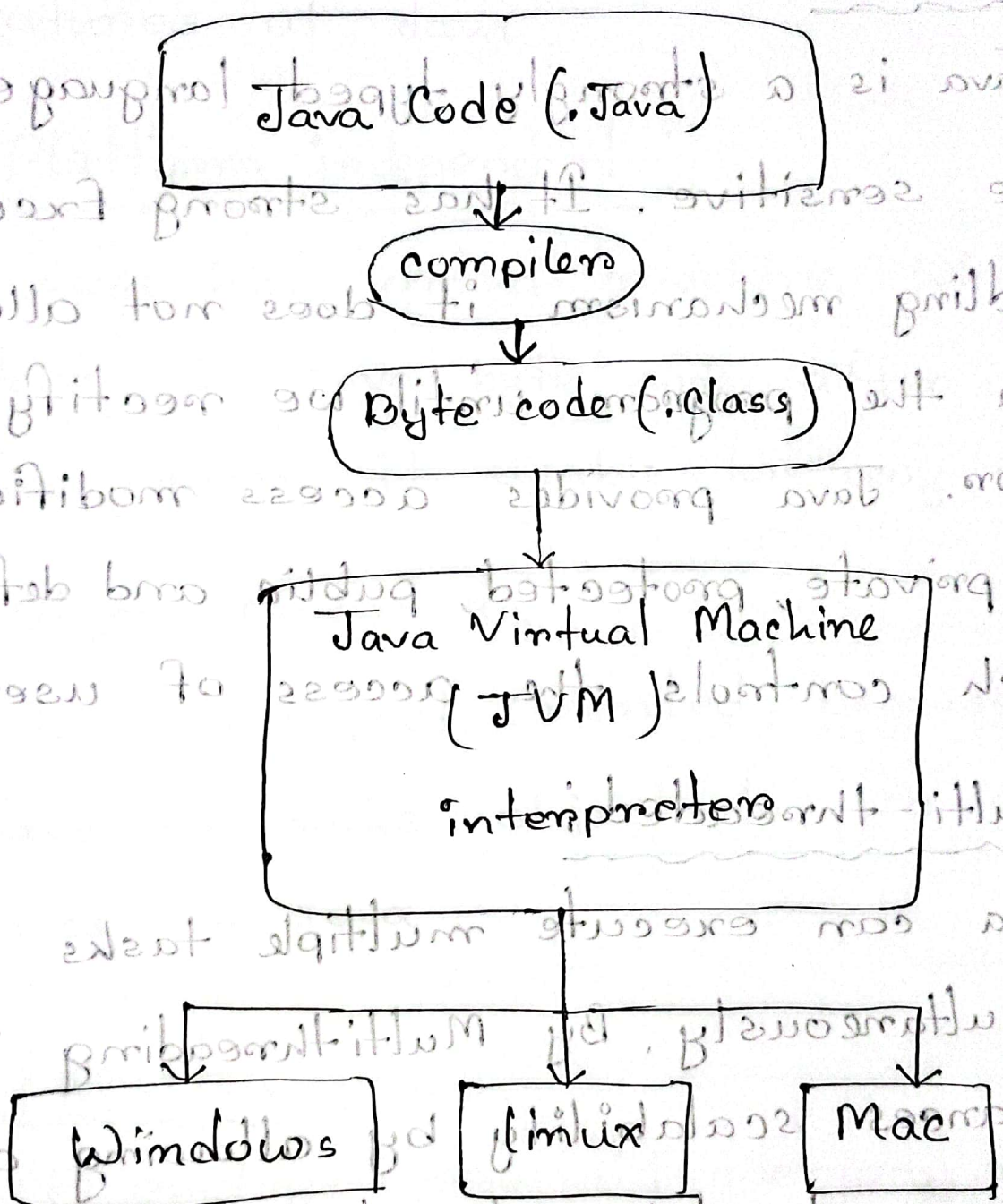
4. Secure :

Java is a strongly typed language and case sensitive. It has strong Exception handling mechanism, it does not allow to run the program until we rectify the error. Java provides access modifier like private, protected, public and default which controls the access of users.

5. Multi-threaded :

Java can execute multiple tasks simultaneously. By Multithreading it enhances scalability by allowing a program to take advantage of multiple CPU cores or processors. Utilization of available hardware resources and improving overall system performance as workload increase.

Java working process :



Object Oriented programming :

* Class :

A class encompasses with variables and methods. We have to write Java code inside a class.

* Object :

It's also called instance. Object means a particular item that belongs to a class. To access any class we have to make an object of that class.

* Everything in Java is treated as belongs to a class.

Three basic principles of OOP:

1. Encapsulation:

Encapsulation is the mechanism that binds together code and the data it manipulates and keeps both safe from outside interference and misuse. A classes variables are hidden from other classes can only be accessed by method of the class in which they are found.

4. Abstruction:

It is a process of hiding certain details and showing only essential information to the user.

2. Inheritance :

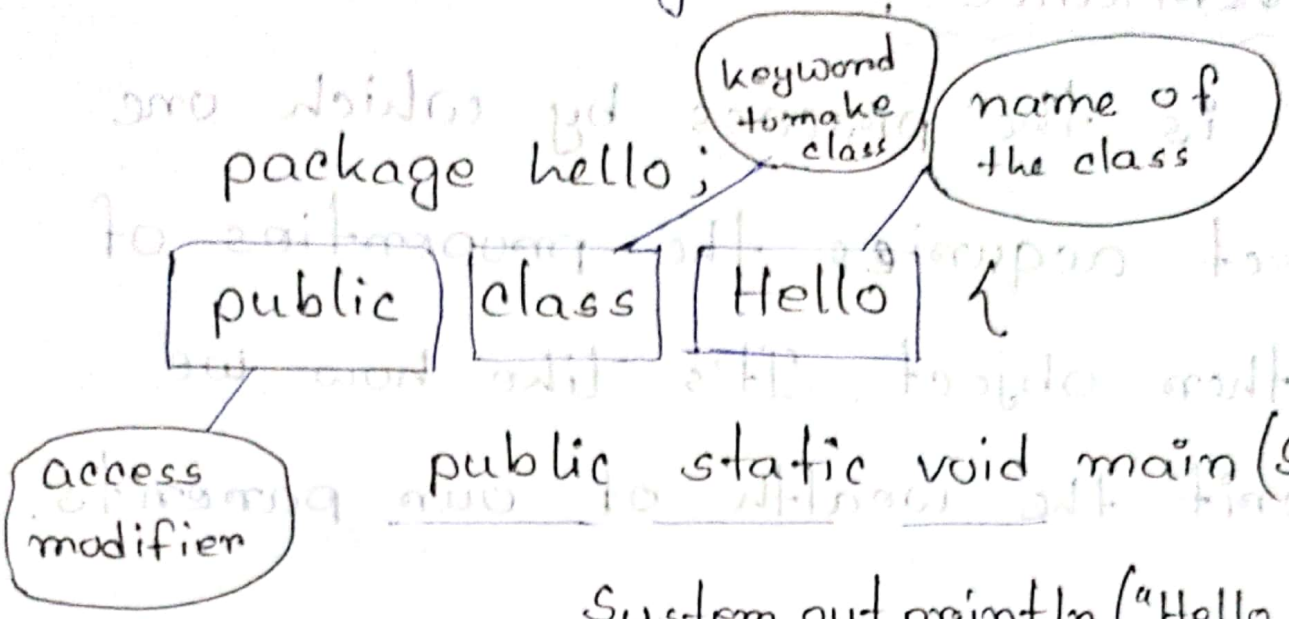
It is the process by which one object acquries the properties of another object. It's like how we inherit the wealth of our parents.

3. Polymorphism :

Polymorphism means ability to have many different form. One ~~inter~~ name interface, multiple methodes. for example in my house I'm the son of my parents but in school I'm a student, if I work in office I'm the employee. Two types polymorphism→

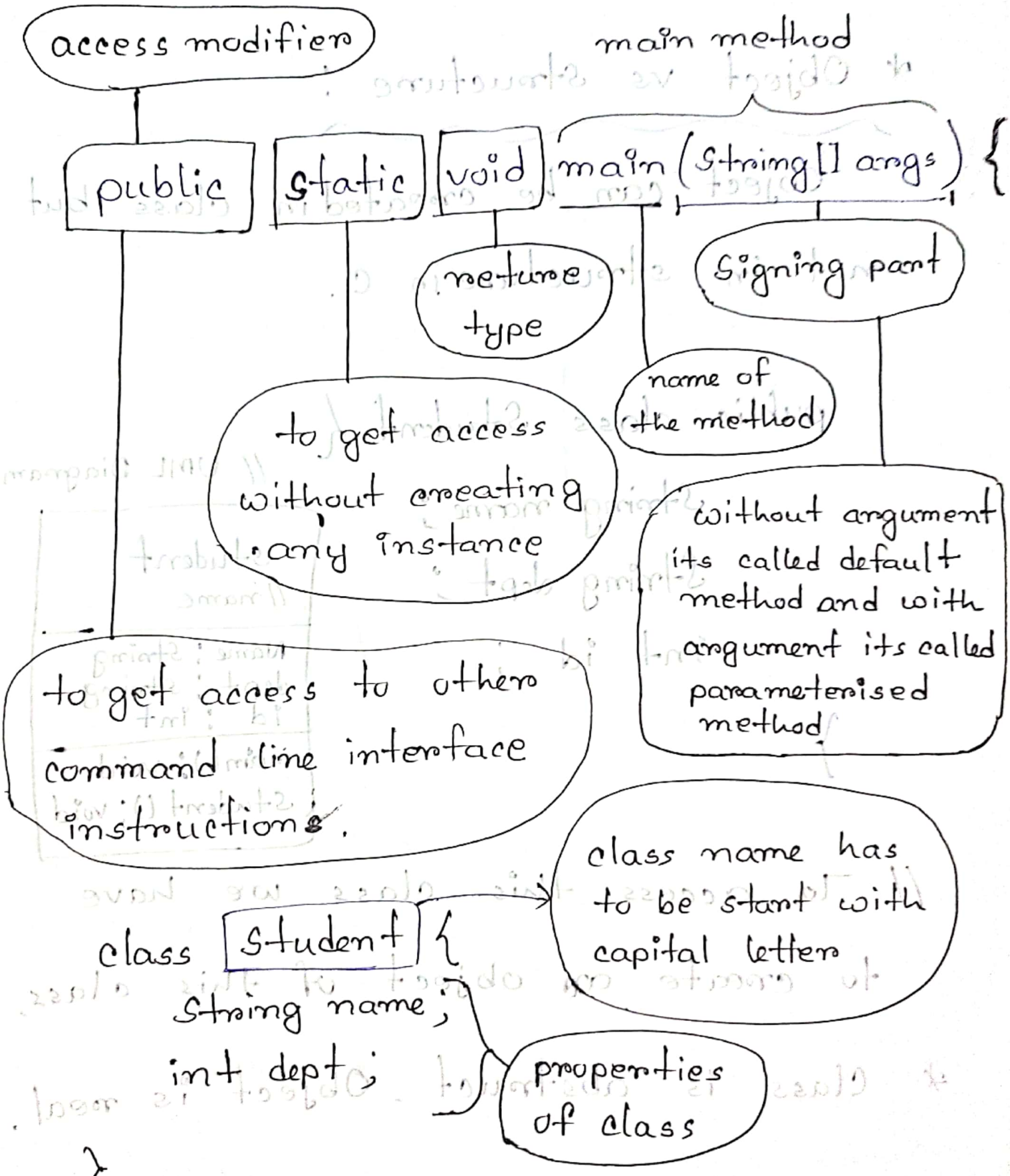
1. compile time polymorphism → method overloading
2. run time polymorphism → method overriding

Simple Java Program :




```
package hello;
public class Hello {
    public static void main(String[] args)
    {
        System.out.println("Hello World");
    } // to print anything this
      method is used.
}
```

* // main method and a class must declare to make code run. The things that must be written to make before any code use is called derivatives.

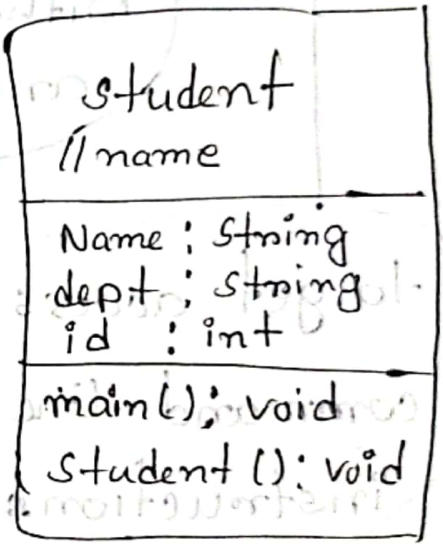

access modifier
main method
public
static
void
main (String[] args) {
return type
Signing part
name of the method
to get access without creating any instance
without argument its called default method and with argument its called parameterised method
to get access to other command line interface instructions.
class name has to be start with capital letter
class Student {
String name;
int dept;
properties of class
}

* Object vs structure :

1. Object can be created in class but not in structure in C.

```
public class Student {
        String name;
        String dept;
        int id;
}
```

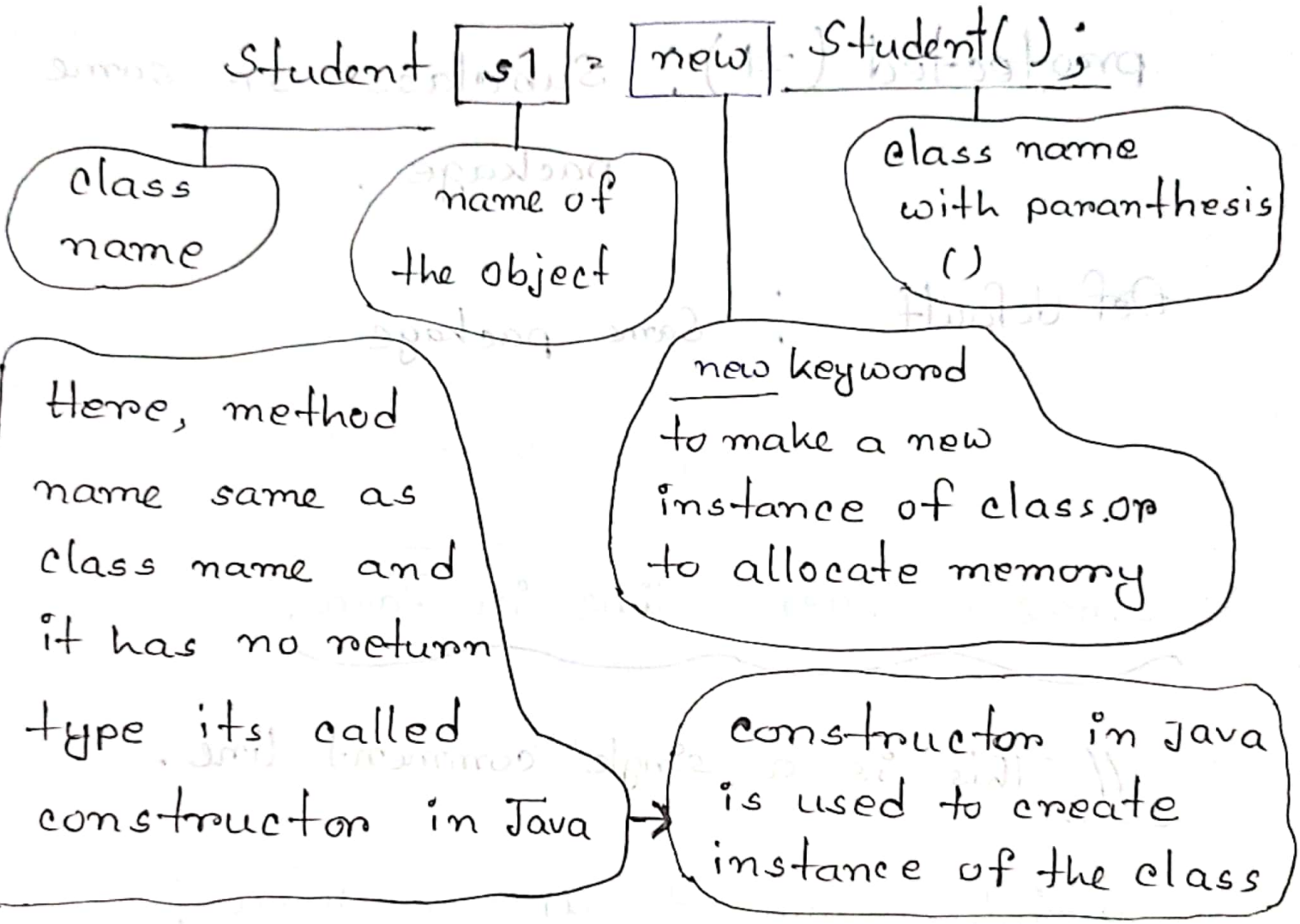
// UML Diagram



// To access this class we have to create an object of this class,

* Class is abstruct. Object is real.

```
public class OOP {

    public static void main (String[] args) {

        Student s1 = new Student();
```




```
        s1.name = "Tamim";
        s1.dept = "CSE";
        s1.id = 23202105;
```

using (.) dot operator we access the class properties.

private (-) : accessable only in same class

public (+) : from anywhere

protected (#) : Subclass or same package.

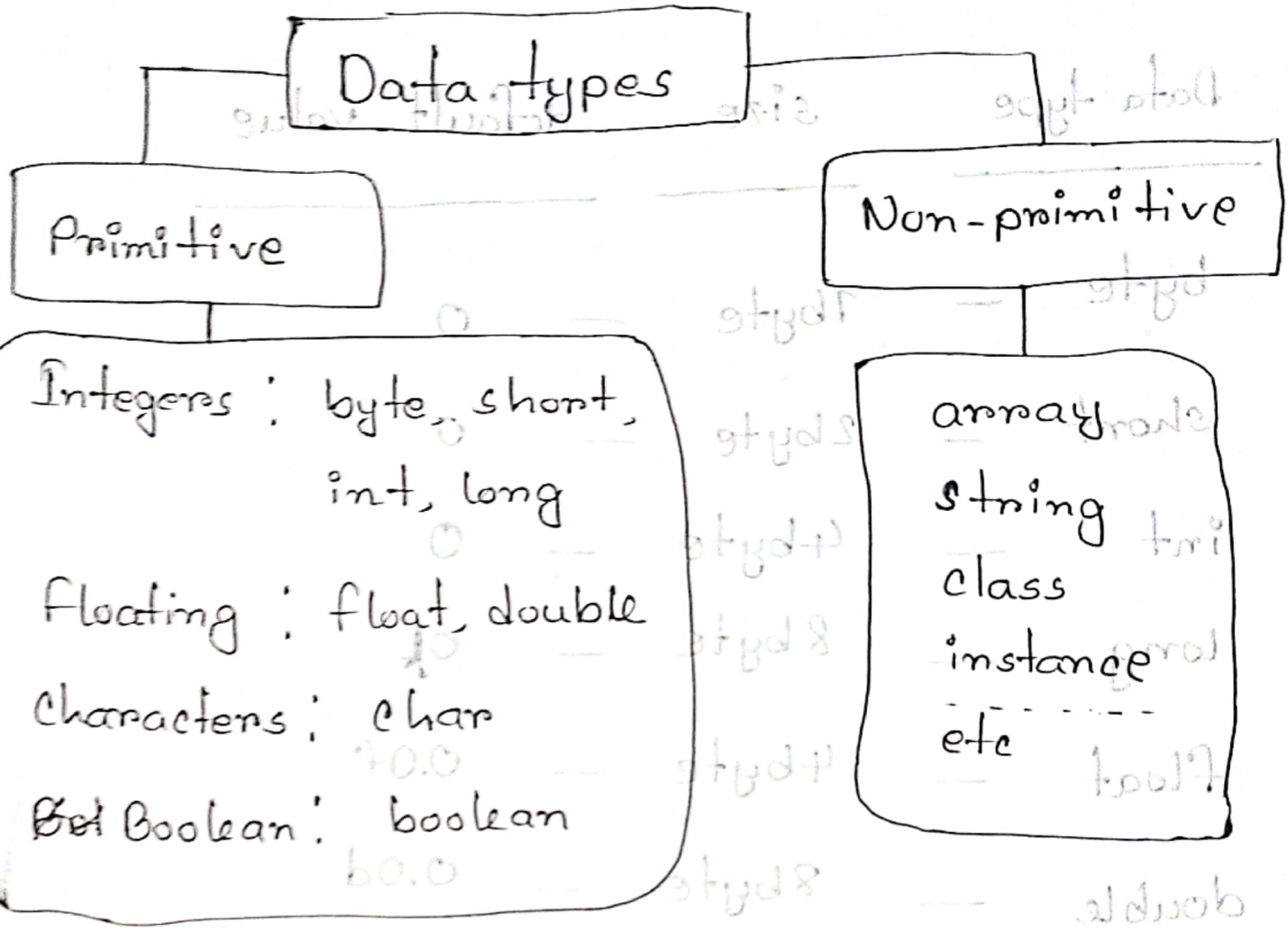
default : Same package

Single comment line in Java:

// This is a single comment line.

Multiple comment line in Java:

/* This is How we write multi-comment line in Java. */



* int i=5;

float f = 5.0f ; // By default compiler treat 5.0 as double value to convert or use it as float we have to put f after value.

Also for, long l = 5001 ;

Char ch = 'a' ; // single qoute for char type value.

Data type	size	default value
byte	1byte	0
short	2byte	0
int	4byte	0
long	8byte	0L
float	4byte	0.0f
double	8byte	0.0d
char	2byte	'\u0000'
boolean	1 bit	false

Operators :

Arithmetic operator :

+, +=, -, -=, *, *=, /, /=, %,

--, ++

Bitwise operator :

NOT - ~

AND - &

OR - |

XOR - ^

SHIFT right - >>

SHIFT left - <<

assignment operator :

= - to assign value

Logical operator:

&&, ||, ==, !=, ?: (Ternary)

Relational operator:

>, <, >=, <=

Type Casting:

* (target type) value

(float) (4*5)/2

Input / Output :

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

// To take input in java we have to use Scanner class. To use Scanner class we have import java.util package inside this the scanner class code is already written. Creating an object of scanner class we can take inputs.

```
class Input {
public static void main() {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    System.out.println(n);
}
}
```

Object name

Object.scanner class method for int data types

methods in Scanner class to take inputs for different data types :

```
int n = sc.nextInt();
float n = sc.nextFloat();
double n = sc.nextDouble();
String n = sc.nextLine();
byte n = sc.nextByte();
short n = sc.nextShort();
long n = sc.nextLong();
boolean n = sc.nextBoolean();
char ch = sc.next().charAt(0);
```

Control Statement:

1. Selection statement
   * if
   * switch
2. Iteration statement
   * while
   * Do - while
   * for
     # for-each
3. Jump statement
   * Break
   * continue

Selection statement:

```
if (condition) {
    // statement 1
} // else if (condition) for multiple condition.
else {
    //statement 2
}
```

Nested if-else:

```
if (    ) {
    if (    ) {
      // code
    }
    else {
      // code
    }
}
else {
}
```

Problem - 1:

// find if the entered number is positive or negetive in Java.

```
package Programing;
import java.util.Scanner;
public class If_else{
    public static void main (String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        if(n>=0) {
            System.out.println("positive");
        }
        else {
            System.out.println("Negative");
        }
}
```

Problem - 2 ;

```
// Check even - odd

package Programing;
import java.util.Scanner;
public class Even_odd {
    public static void main (String[] args){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        if(n%2==0){
            System.out.println("Even");
        }
        else{
            System.out.println("odd");
        }
```

Problem – 3 :

// RGB : color code using switch case.

```
package Programing;
import java.util.Scanner;
public class RGB {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int r = sc.nextInt();
        int g = sc.nextInt();
        int b = sc.nextInt();
        String rgb = r + " " + g + " " + b;
        switch(rgb) {
            case "100":
                System.out.println("Red");
                break;
            case "010":
                System.out.println("Green");
                break;
```

```
        case "001":
            System.out.println("Blue");
            break;
        case "000":
            System.out.println("Black");
            break;
        case "111":
            System.out.println("White");
            break;
        case "110":
            System.out.println("Yellow");
            break;
        default:
            System.out.println("Invalid!");
        }
    }
}
```

Iteration statement :

for loop :

```
for (initialization; condition; updation){
    // body
}
```

example :

```
for (int i=0; i<n; i++){
    System.out.println(i);
}
```

for-each loop :

```
for (datatype variablename : collection/value)
{
  // body
}
```

example :

```
int num[] = {10, 20, 30};
for (int i : num) {
    sum += i;
}
```

While loop :

```
initialization
while (condition) {
    // body
    updatation
}
```

example :

```
int n = 5;
while (n >= 0) {
    System.out.println(n);
    n--;
}
```

do-while loop:

```
initialization
do {
  // body
  // updation
} while (condition);
```

example:

```
int i = 5;
do {
  System.out.println(i); i--;
} while (i > 0);
```

Problem - 1 :

// even number print

```
package Programing;
import java.util.Scanner;
public class Even {

public static void main (String [] args){

    Scanner sc = new Scanner (System.in);

  int n;

  n = sc.nextInt();

 for(int i=1;  i<=n; i++){

      if ( i%2 ==0) {

        System.out.println(i);

        }

    }

}
```

Problem - 2 :

// Odd number print

```
package Programing;
import java.util.Scanner;
public class Odd {
    public static void main (String [] args) {
        Scanner sc = new Scanner (System.in);
        int n = sc.nextInt();
        for(int i=1; i<=n; i++){
            if (i%2 != 0) {
                System.out.println(i);
            }
        }
    }
```

Problem - 3 :

// Find factorial of a given number

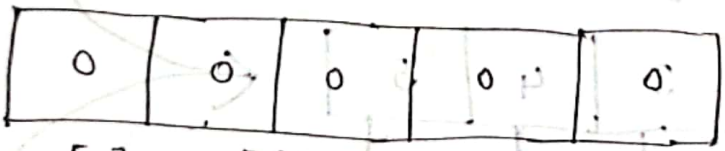
```
package Programming;
import java.util.Scanner;
public class factorial {
    public static void main (String [] args){
        Scanner sc = new Scanner (System.in);
        int n = sc.nextInt();
        int fact = 1;
    for( int i = 1; i <= n; i++) {
            fact *= i;
    }
    System.out.println ("factorian: " + fact );
}
}
```

Array :

Array is a collection of similar data types values stored at contiguous memory locations.

Insted of declaring 100 values for same purpose age1, age2, age3 ---- .
We can use array.

int arr[5] → Index of array starts from 0, and



* Normal variable `int d = 2` can't access memory directly we have to create object with the help of constructor. then with object reference memory will be allocated

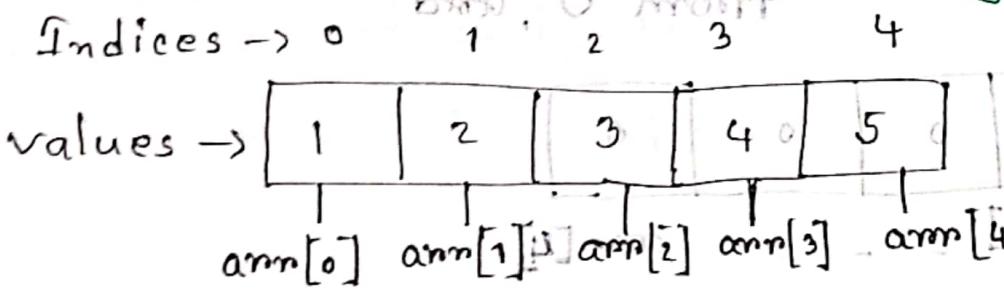
Array declaration :

```
dataType[] arrayname;
int[] arr;
int arr[];
int arr[] = new int[5]; // size of array
```

```
int i=20;
```
doesn't access memory directly

```
int[] age = new int[5]; //(if we initialize this way directly access memory with default value 0)
int[] arr = {1,2,3,4,5}
```

// direct access to memory without creating object.



directly access the memory and put the values in a certain location

```
class Array {
    public static void main(String [] args) {
        int [] age = new int[5];

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

Access array using loop :

```
class Array {
    public static void main(String [] args){
        int [] age = new int[5];
    for(int i=0; i<5; i++){
        System.out.print(age[i]+" ");
        }
    }
}
```

Access array using for-each loop:

```
class Array {
    public static void main (String [] args) {
        int [] age = {12, 4, 5, 2, 5};
        for (int i : age) {
            System.out.println (i);
        }
    }
}
```

Problem - 1:

// Write a program which computes sum and average of values stored in an array.

```
package Programing;
import .java.util.Scanner;
public static void main (String[] args){
    Scanner sc = new Scanner(System.in);
   int [] arr = {1, 2, 6, 10, 12};
   int sum = 0;
   float avg;
   for (int i : arr){
          sum += i;
   }
   avg = ((float) sum/(float) arr.length);
   System.out.println("Sum :" + sum);
   System.out.println("average :" + avg);
 }
}
```

Problem - 2 :

// find the largest element in an array?

```
package Programing;
public static void main (String[] args){
    int [] arr = { 1, 3, 51, 12, 13 };
    int max = arr [0];
    for (int i : arr) {
        if (max < i ) {
            max = i ;
        }
    }
    System.out.println ("Largest :" + max );
}
}
```

Problem -3 :

// Write a java program to check if an array contains a given value ?

```
package programing;
import java.util.Scanner;
public static void main(String[] args){
    int[] arr = {1, 3, 5, 2, 11};
    int value;
    Scanner sc = new Scanner(System.in);
    int value = sc.nextInt();
    int count = 0;
    for (int i : arr){
        if ( i == value){
            System.out.println("Value found at"+ i);
            count ++;
            break;
        }
    }
    if (count == 0) System.out.println("Not found");
}
```

Multidimensional array :

2D array :

Declaration :

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

	0	1	2	3
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

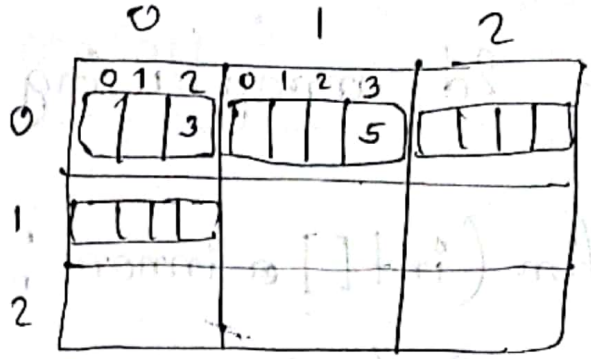
```
int[][] a = { {1,2,3},
              {4,5,6,9},
              {7} };

for(int i=0; i<a.length; i++){
    for(int j=0; j<a[i].length; j++){
        System.out.println(a[i][j]);
    }
}
```

access 2d array using for-each loop;

```
for (int[] inner : a) {
    for (int data : inner) {
        System.out.println (data);
    }
}
```

// a[0][0][2] = 3

3d array :



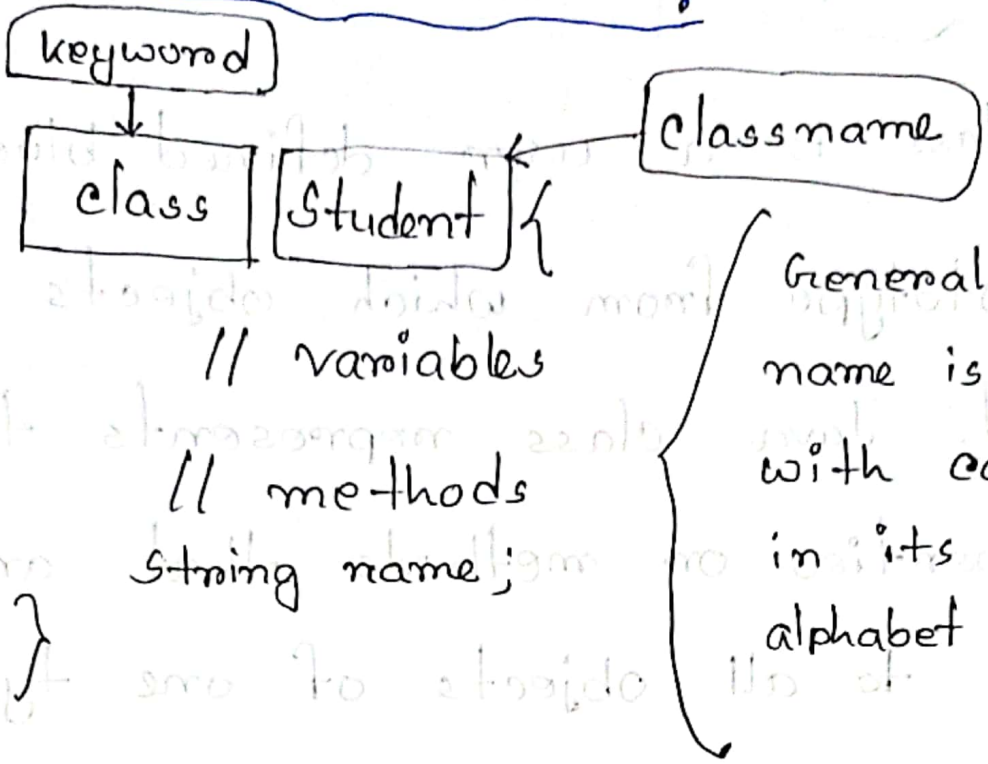
declaration :

```
int [][][] test = { { {1,2,3}, {2,3,4}},
                    {{-4,-5,6,9},
                     {1},
                     {2,3}}
                  };

for( int[][] array2D : test ) {
    for (int [] array1D : array2D) {
        for( int item : array1D) {
            System.out.println (item);
        }
    }
}
}
```

Java class :

A class is a user defined blueprint or prototype from which objects are created. Java class represents the set of properties or methods that are common to all objects of one type. In Java we have to declare at least one class. We can make more than one class in Java. When a set of classes grouped together are known as package in Java. Inside class we write variable, methods.

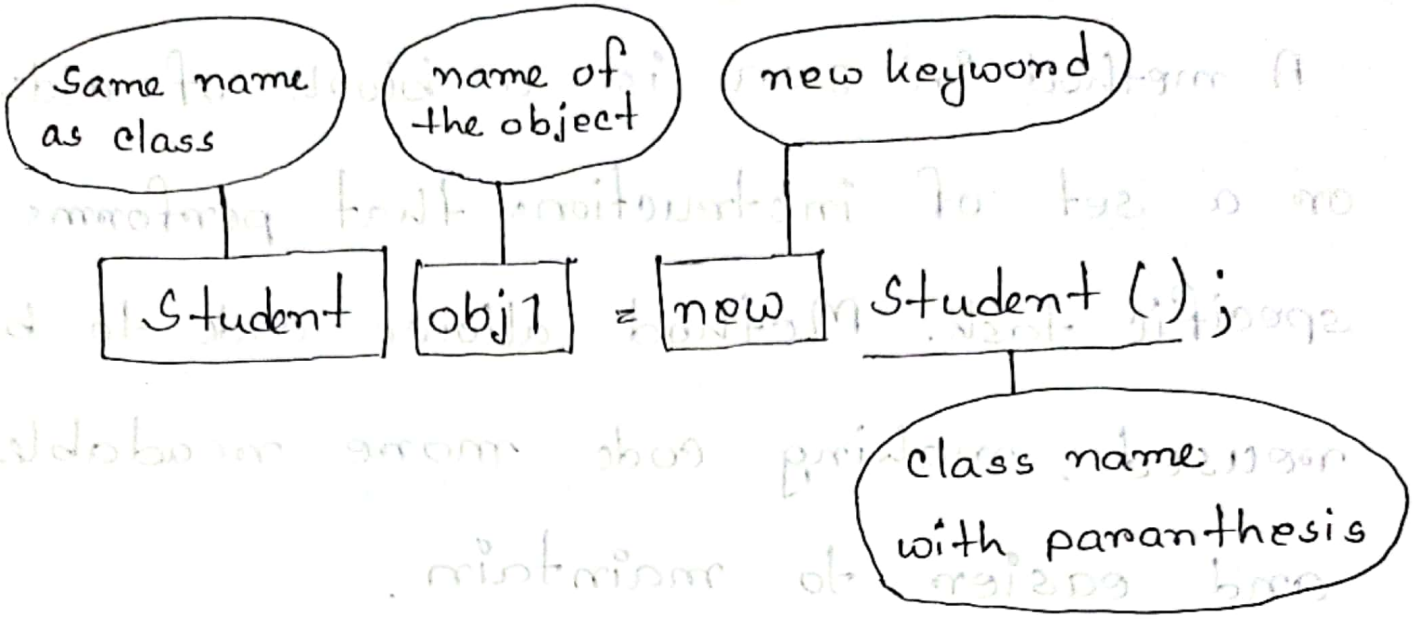
Structure of class :



Java objects :

To access any class we have to create an object of that class. Using the object reference we can access any variable or methods of that class.

Structure of object :



Example :

```
public class Student
```

* Accessing class variables, methods by (.) dot operator :

```
obj1.name = "Tamim";
```

Methods in Java:

A method in Java is a block of code or a set of instructions that performs a specific task. Method allows code to be reused, making code more readable and easier to maintain.

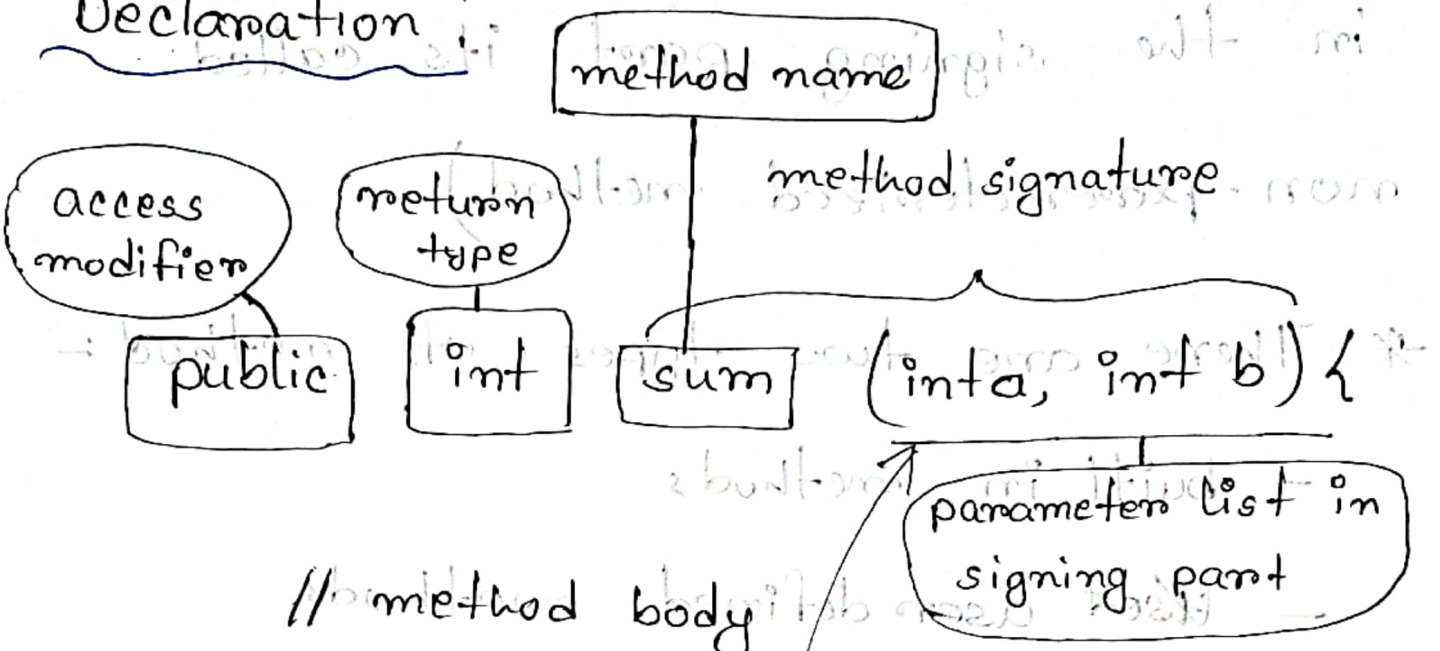
* Method has a return type (which data type it will return when it called)
* We have to call method to execute or perform the code inside it.
* Method can be parameterized or non-parameterized. (if we pass some values in the calling signing part its called parameterized method and if we don't pass any argument (value)

in the signing part. its called non-parameterized method).

* There are two types of method :-
  - built in methods
  - ~~User~~ user defined methods

The methods we made for by us, its called the user defined method. We have to define the body of the method. But builtin method are already defined in the class libraries : We just call the the method to use it. But for some method we have to import the class libraries to use it.

Declaration :



method calling :

value is pass to the formal parameter of the method 10, 20 will be copied in a, b

sum (10, 20);

// argument / ~~formal~~ actual parameter

* if the method is defined in another class then we have to access the method through the object of that class.

Example :

```
package programing;
class Student {
    void printinfo (int i , String s){

        System.out.println ("ID :" + i );
        System.out.println ("Name :" + s );
    }
}
public class Method {
  public static void main (String [] args ) {
        Student obj = new Student ( );
        obj. printinfo ( 232002105 , "Tamim");
      }
  }
```

Constructor in Java:

Constructor is a special type of method used to initialize instance of a class.

* It has no return type.
* name of the constructor is same as class name
* We don't have to call it like method it calls itself at the initialized time.
* We can send or pass argument in the constructor. it is parameterized constructor.
* Without any argument it's default constructor.

Declaration :

```
constructor_name() {
    // body
}
```

or,

```
constructor_name(parameter) {
    // body
}
```

Example :

```
package programing;
class Student {
    Student (int id, String name){
        System.out.println (id);
        System.out.println(name);
    }
}
public class Method {
    public static void main(String[] args) {
        Student obj = new Student (232, "Tamim");
    }
}
```

// We don't need to call the constructor it automatically calls when we create ~~obj of~~ object of that class.

Types of variable in Java :

1. Local variable :

A variable declared inside the body of the method is called local variable. This variable can only be used in that method. A local variable doesn't even exist in another method.

2. Instance variable :

A variable declared inside the class but outside the body of the method is called an instance variable. To access instance variable we have to create an instance of that class.

3. Static variable :

Static variables are created with static keyword in a class. No matter how many object is created of that class in every object static variable will be the same copy of the initizied one. To access static variable of other class we don't have to create an instance of that variable. We can access with only class name.

* We can't use non-static variable directly in a static method. But with object reference we can use non-static variables.

Example :

Local variable :

```
class Student {

    void show() {
        String name = "T";
        int id;
    }
}
// variable doesn't exist outside
   the method.
```

Instance variable :

```
public class Student {

    public String name;
    // public instance variable

    private int age;
    // private instance variable

    String cg;
    // default instance variable

}
public class Main{
public static void main (String [] args){
    Student obj = new Student (System.in);

    obj.name ;

    obj.cg ;

  }
}
```

Static variable :

```
class Student {
    String name;
    int id;
    static String dept = "CSE";

    Student (String name, int id) {
        this.name = name;
        this.id = id;
    }
    void print () {
        System.out.println (name + ", " + id + ", " +
                            dept);
    }
}

public class Main {
psvm () {
    System.out.println (Student.dept);
    // (without object access through
        class name)
    Student obj1 = new Student ("Tamim", 105);
    obj1.print ();
} }
```

Difference between Methods & Constructor :

Method	Constructor
1. Method can be any user defined name	1. Constructor must be class name.
2. Method has a return type.	2. Constructor has no return type.
3. Method should be called explicitly either with object reference or class reference.	3. Constructor will be called automatically whenever object is created.
4. Method is not provided by compiler in any case.	4. Java compiler provides a default constructor if we don't have any constructor.
5. It is used to show behaviour of an object.	5. It is used to initialize the state of an object.

Method Overloading :

If a class has multiple methods having same name but different in parameters, it is known as method overloading.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

* There are two ways to overload the method in Java :

1. By changing number of arguments.
2. By changing the data type.

Constructor overloading is also possible it's the same as method overloading.

Examples :

Method overloading by changing no. of arguments :

```
package Programing;
class Calculator {
    public int sum (int a, int b) {
        return a+b; // 3
    }
    public int sum (int a, int b, int c) {
        return a+b+c;
                // 1 + 2 + 3
    }
}
public class Overloading {
    public static void main (String[] args) {
        Calculator c1 = new Calculator();
        c1.sum (1, 2);
        c1.sum (1, 2, 3);
    }
}
```

which method will get called depends on the number of argument we pass when we are calling

Method Overloading by changing data types:

```
package Programing;
class Calculator {
    public int sum(int a, int b) {
        return a+b; // 14
    }
    public double sum(double a, int b) {
        return a+b; // 8.7
    }
}
public class Moverloading {
    public static void main(String []args){
        Calculator c = new Calculator();
        int add1 = c.sum(7,7);
        double add2 = c.sum(7.7, 1);
    }
}
```

// when we pass double type value
double type method gets called.

Call by value :

1. If we call a method by passing a primitive data type value it is known as call by value.

2. The value is copied to a method parameter.

3. Changes to that formal parameter doesn't affect the actual parameter.

4. In call by value original value doesn't change.

Example :

```
package programing;

class CallByValue {

    void change(int i) {  // formal parameter
        i = 20;
    }  // changing the value of i = 20.
}      // doesn't change the actual value of x.

public class Test {

    public static void main(String[] args) {

        CallByValue obj = new CallByValue();

        int x = 10;

        System.out.println(x);  // output : 10

        obj.change(x);  // Here x is actual parameter

        System.out.println(x);  // output : 10

    }
}
```

i & x are two different variable there is no conection between them.

10

i is the local variable to this method. Here simply we pass the value to a formal parameter i copy the value of x

10

Call by reference :

1. If we call a method by passing a non-primitive / reference type data (object, string etc) then it is known as call by reference.
2. Changes to that formal parameter does affect the actual parameter.
3. In call by reference original value gets change.

Example :

```
package OOP;

class CallByRef {

    String name;

    void change(CallByRef r2) {
        r2.name = "Tamim";
    } // changes through reference affect the original.
}

public class Test {

    public static void main(String[] args) {

        CallByRef r1 = new CallByRef();

        r1.name = "Taslim";

        System.out.println(r1.name); //Taslim

        r1.change(r1); // Insted of passing the value we pass the object which is the reference variable

        System.out.println(r1.name);

        // output: Tamim

    }
}
```

Instead of data type and variable here we get class name & object because we pass object and this object belongs to CallByRef class.

// copy of reference is passed. But both the original & copied reference point to the same object in memory.

Constructor Overloading :

```
class Time {
    private int h,m,s;

    Time() {
        this.h = 10;
        this.m = 30;
        this.s = 55;
    }

    Time(int hour, int min, int sec){
        this.h = hour;
        this.m = min;
        this.s = sec;
        timeshow();
    }
```

```
Time ( Time t ) {
        this.h = t.h;
        this.m = t.m;
        this.s = t.s;
    }
    void timeshow () {
        System.out.println (h + ":" + m + ":" + s);
    }
}
public class TimeExample {
    PSVM () {
        Time t1 = new Time ();
        t1.timeshow ();
        Time t2 = new Time (10, 20, 30);
        Time t3 = new Time (t2);
        t3.timeshow ();
    }
}
```

Variable-length argument :

Variable Arguments (~~vararg~~ Varargs) in Java is a method that takes a variable number of arguments. If we don't know how many argument we will have to pass in the method then we should use varargs. In method normally the parameters are fixed the number of formal parameters decides how many arguments we can pass and work with it. Using varargs, it is more dynamic the parameters method can recive is not fixed.

Syntax:

number of arguments

```
public static void fun (int ... a){
    // method body
}
```

the concept of array is implemented Here. We can access the argument value with index

Example:

```
public class Argument {
    static int sum (int ... arg){
        int r = 0;  // int[] arg;
        for (int i=0; i<arg.length; i++){
            r += arg[i];
        }
        return result; } }
    public static void main (String[] args){
        System.out.println (sum(1,2)); // 3
        System.out.println (sum (1,2,3,4)); // 10
        System.out.println (sum()); // 0
    }
}
```

Recursion :

When a method calls itself then the method is known as recursive method. And the whole process of this method calling itself again and again is known as Recursion.

Example :

```
public class factorial {
    int fact (int n) {
        int result = 0;
        if (n==1) return 1;
        // Base case to stop recursion
        result = fact(n-1)*n; // general case
        return result;
    }
}
```

function calling itself.

```
import java.util.Scanner;
public class NewClass {
    public static void main(String[] args) {
        factorial f = new factorial();
        int n, r;
        Scanner sc = new Scanner(System.in);
        n = s.nextInt();
        r = f.fact(n); // ~~fc~~ calling method
        System.out.println(r);
    }
}
```

Java Inheritance

Inheritance in one of the key features of OOP. that allows us to define a class from an existing class.

It is a mechanism where one class called a subclass / child class inherits properties and behaviors from another class called a superclass / parent class. Here only child can access both his and parents properties. Parents has the access to his properties but not his child properties. We use extends keyword to inherit from a class, Main objective is to Re-use of that code.

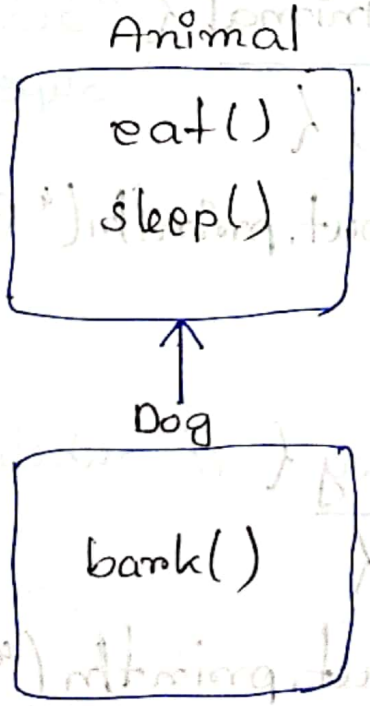
Syntex :

```
class Animal {
    // eat() method
    // sleep() method
}
class Dog extends Animal {
    // bark() method
}
```

Superclass / parent class

subclass / child class

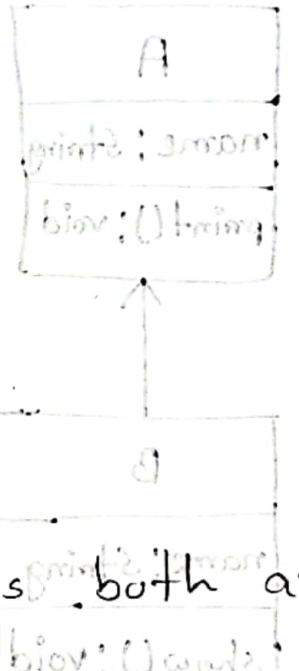
// child class can access Animal classes method, variables.



Example:

```
package OOP;

class Animal {   // Superclass
    void eat() {
        System.out.println("I can eat");
    }
    void sleep() {
        System.out.println("I can sleep");
    }
}

class Dog extends Animal {  // Subclass of Animal
                            // superclass of Cat
    void bark() {
        System.out.println("Bark Bark");
    }
}

class Cat extends Dog {  // Subclass
    void mew() {
        System.out.println("mew mew");
    }
}
```

```
public class Inheritance {
    public static void main (String [] args){
        Cat obj = new Cat();
        Obj.mew();
        obj.bark();
        obj.sleep();
        obj.eat();
    }
}
```

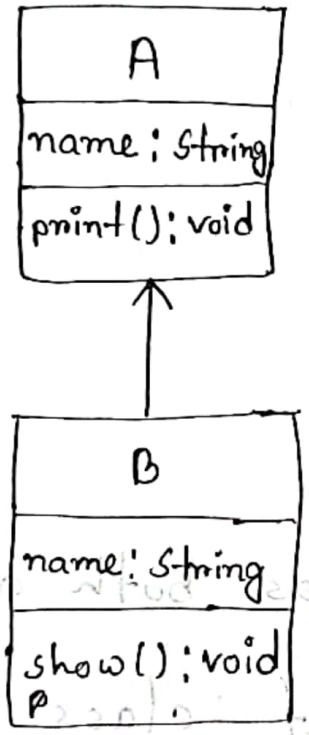


* Animal & Dog class both are the parent class of cat class so cat class can access both classes properties.

Type of Inheritance :

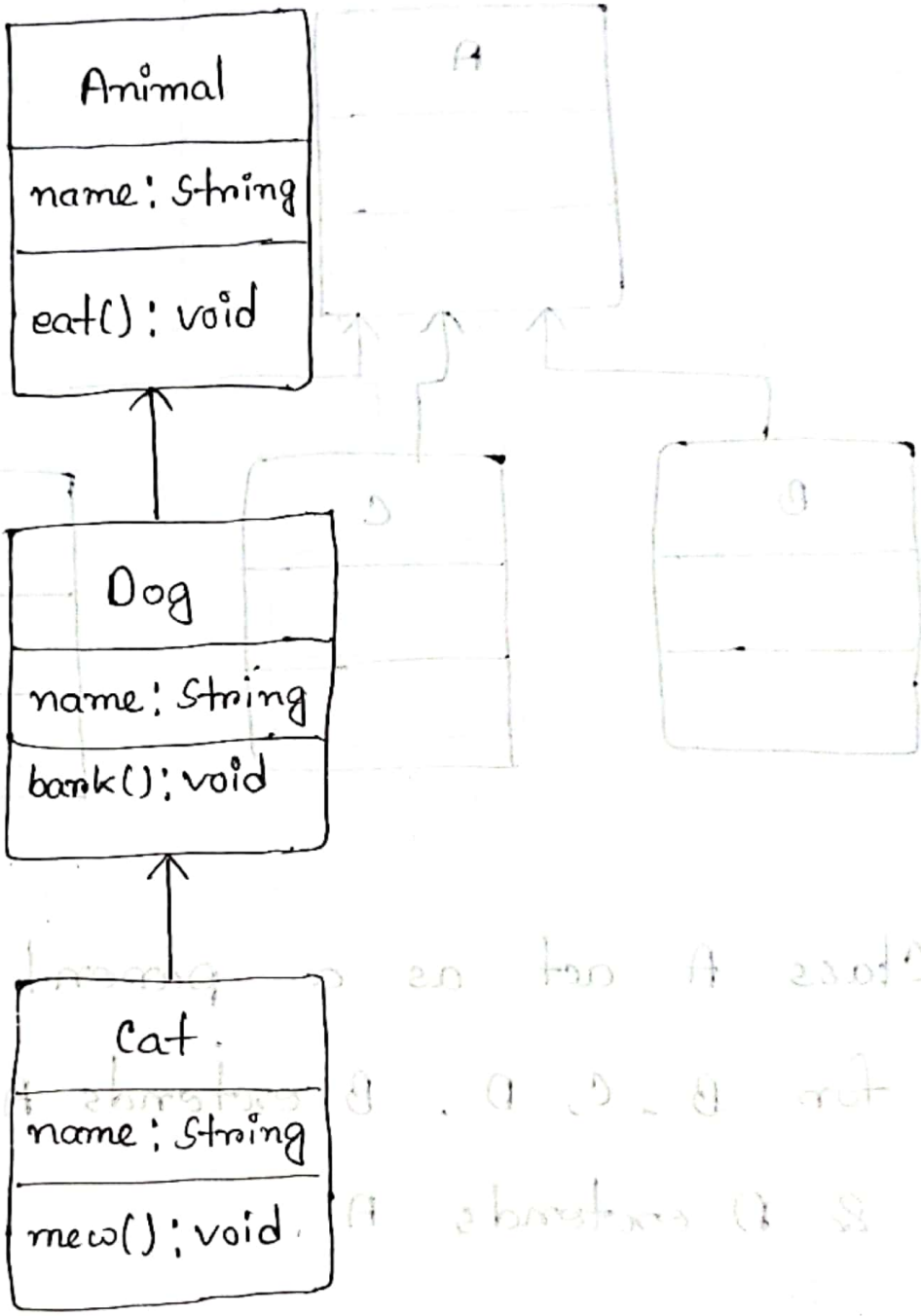
There are five types of Inheritance :

1. Single inheritance :



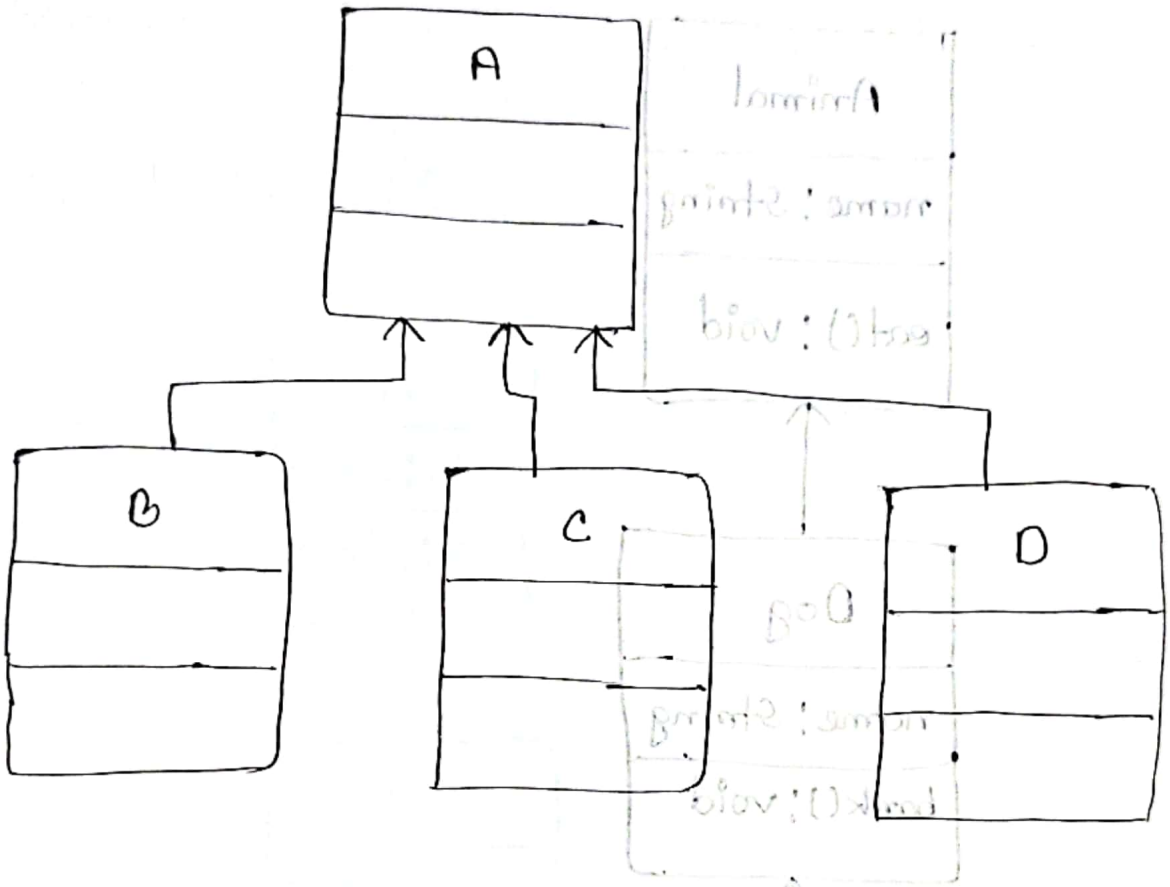
class B extends from class A.

2. Multilevel inheritance :



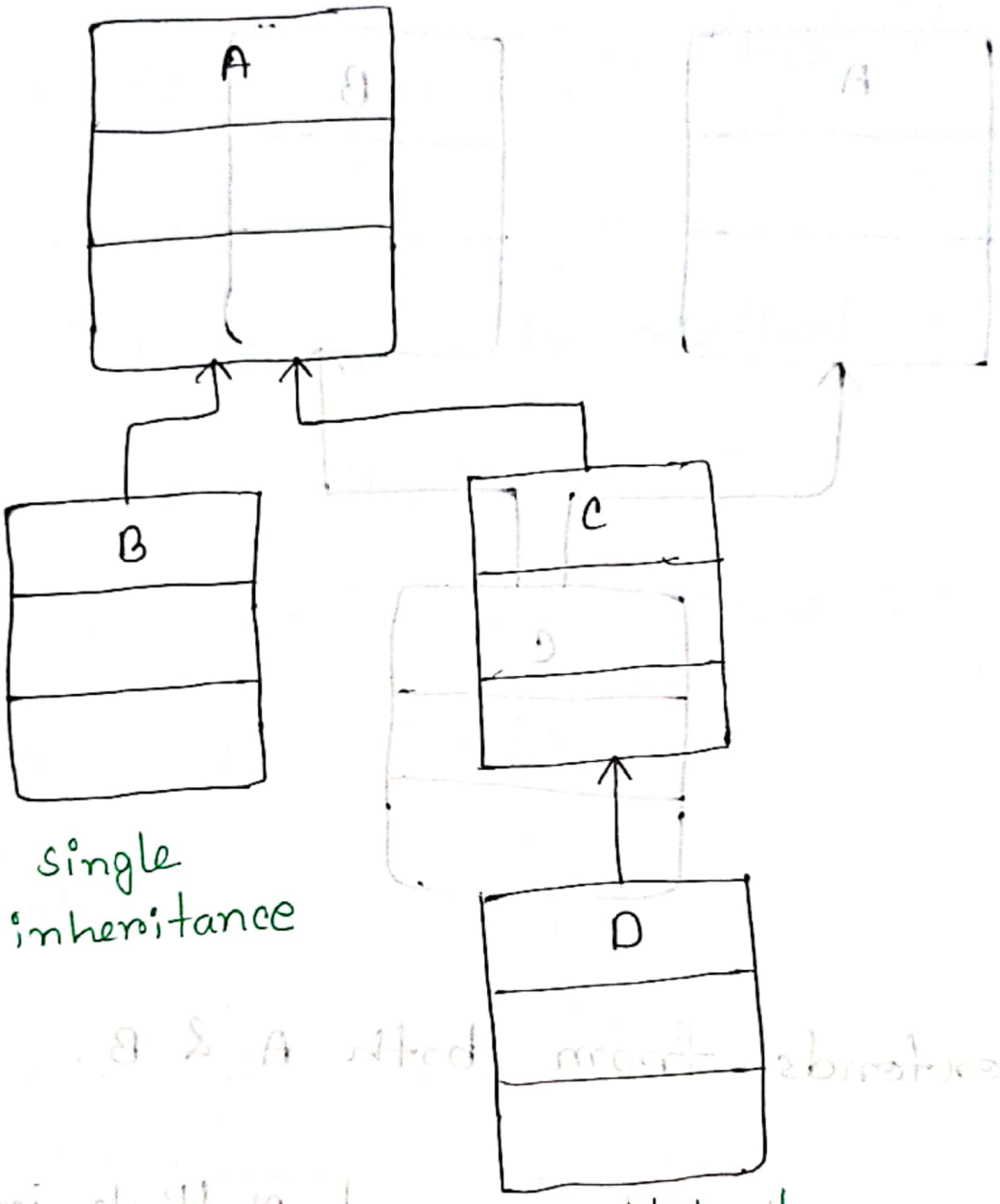
cat extends Dog & Dog extends Animal

3. Hierarchical inheritance :



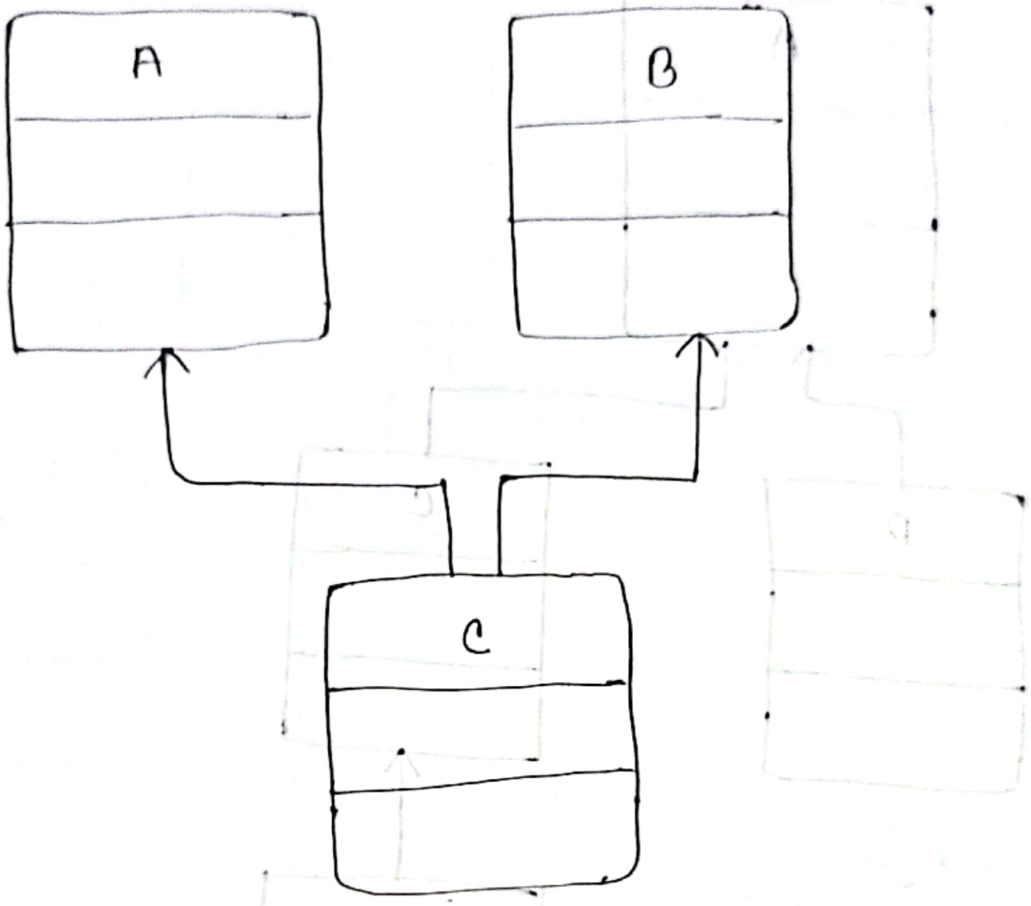
Class A act as a parent class for B, C, D. B extends A, C extends A & D extends A

4. Hybrid inheritance



Mix of two or more types of inheritance.

5. Multiple inheritance :



C extends from both A & B.

(But Java doesn't support Multiple inheritance)

* Multiple inheritance can be achieve through interface.

Method overriding :

When same method is defined in both superclass and subclass. We call the method from main method then the subclass method overrides superclass method. And the subclass methods gets executed this is known as method overriding :

* Using the super reference A parent method can be invoked
* If a method is declared with final modifier it can't be overridden

Example :

```
package oop;
class Animal {
    public void eat(){
        System.out.println("I eat");
    }
}
class Dog extends Animal{
    public void eat(){
        System.out.println("Dog eat");
    } // child class method overrides
}
public class Main{
    public static void main(String[] args){
        Dog d = new Dog();
        d.eat();//'Dog eat' will print.
    }
}
```

Overloading vs Overriding

Method Overloading	Method Overriding
Method overloading is compile time polymorphism	Method Overriding is run time polymorphism
It occours within the class.	It occours in two classes with inheritance relationship.
Methods must have same name and different signatures	Methods must have same name and same signature
private and final methods can be over-load	private and final methods can't be over-riden
It helps to increase the readability of a program.	It allows the subclass to change the behaviour of a method from the parent class.

Encapsulation :

Encapsulation is the mechanism that binds together code and the data it manipulates and keeps both safe from the outside interference and misuse. A class variables are hidden from other classes and can only be accessed by method of that class. With the method we accessed the private or encapsulated data is called getter and setter.

getter method is used to read or accessed or get the encapsulated data.

setter method is used to write or set new data.

Example :

Getter :

```
package oop;                          // Private variable can't be accessed from another class

class Encap {
    private int id = 232002105;
    public int getId(){ // getter method
        return id;                    // returns the value of id
    }
}
public class Getters {
    public static void main(String[] args){
        Encap obj = new Encap();
        System.out.println(obj.getId());
```

Example :

Setter :

```
package OOP;
class Human {
    private int age = 18; // 22
    public void setAge(int a) {
        age = a;
    }
}
public class Setters {
    public static void main(String[] args) {
        Human obj = new Human();
        obj.setAge(22);
        System.out.println(Obj.getAge());
    }
}
```

22

// using method we access the private variable then change or update value.

Abstract:

An abstract class in Java is a class that cannot be instantied and designed to be inherited by other classes.

An abstract class can have both abstract methods (methods without body) and concrete methods (with body).

Abstract class: A class that contains at least one abstract method and cannot be instantiated.

Abstract method: A method declared without body, ment to be implemented by subclass.

Example:

```
package OOP;

abstract class Animal { // we can't create object of abstract class.
    void eat() {
        System.out.println("I eat");
    }

    abstract void sleep(); // abstract. methods body is defined in the subclass.
}

class Dog extends Animal { // to access abstract class it has to be inherited.
    void sleep() {
        System.out.println(""); // body of abstract method must be defined in subclass otherwise we'll get error.
    }
```

Some Important points :

1. An abstract class must be declared with an abstract keyword.
2. It can have both abstract and non-abstract method.
3. It cannot be instantiated.
4. It can have final method which will force the subclass not to change the body of the method.

Abstraction in Java :

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

There are two ways to achieve abstraction in Java:

1. Abstract class
2. Interface

* By using Interface we can achieve 100% of abstraction.

Interface in Java :

Interface is just like a class, which contains only abstract method. And all the abstract methods of interface must be overridden by an inherited class.

If we declare a method inside interface by default it becomes **public & abstract**.

Also the variable declared in the interface is **public + static + final** by default.

* When a variable is declared with **static** that variable takes memory without creating any objects.
* final : when a variable is declared with **final keyword** we have to set a final value of it and we can't change it.

Declaration :

keyword to create interface

```
interface A {
    void show(); // By deafalt :
    // public abstract void show();
}
```

// abstract method must be defined by inherited class. Interface is like abstract we can't create object of it.

```
class B implements A {
    public void show() {
        System.out.println("Show");
    }
}
```

implements keyword is used to inherite the interface

// we have to implement all the abstract method otherwise that class becomes abstract class.

```
    public static void main (String [] args) {
        B obj = new B();
        B.show();
    }
}
```

```
interface A {
    int age = 20;
}

class B implements A {
    psvm( ) {
sout(A.age);
    }
}
```

// if we create varible by default it becomes {public static final} So, we have to set a value in initialization time. And the value is final we can't change it.

we can call the variable directly using interface name cause it is static. No need to create object. static varibles takes memory without object creation

```
interface A {
    // code
}

interface B extends A {
    // code
}

class C implements B {
    // code
}
```

* when interface inherit another interface we have to use extends keyword.
* when interface is inherited by another class implements keyword is used.

Example :

```
interface Polygon {
        void getArea();
}
class Rectangle implements Polygon {
    public void getArea() {
        int length = 5;
        int breadth = 6;
        System.out.println("Area: " + length * breadth);
    }
}
class Main {
    public static void main(String[] args) {
        Rectangle r1 = new Rectangle();
        r1.getArea();
    }
}
```

Multiple inheritance in Java:

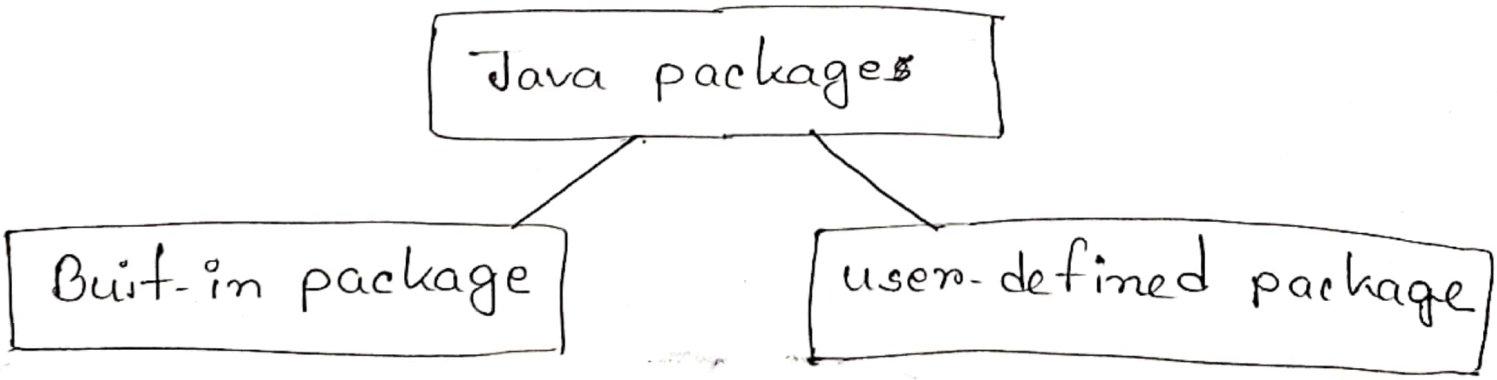
```
interface A {
        void show();
}
interface B {
        void display();
}
class C implements A,B {
        public void show() {
                sout("  ");
        }
        public void display() {
                sout("  ");
        }
} // class C has the property of both
    A & B interfaces it's called multiple
    inheritance.
```

Abstract class Vs. Interface :

Interface	Abstract class
1. Can have only abstract methods.	1. can have abstract & non abstract methods
2. Support's multiple inheritance.	2. doesn't support multiple inheritance.
3. has only static and final variable.	3. Can have static, non-static, final & non-final variables.
4. fully abstract.	4. partial abstraction.

Package in Java :

Package in java is same as folder in windows. In java it is a mechanism to encapsulate a group of similer classes, sub-packages & interface. When we need a particular class defined in that package. We can import the package and use it classes, interfaces in another packages.



Built-in packages :

some of the packages are already defined by the developer. We just import the package and use it.

* Some of the commonly used built-in packages :

1. Java.lang :

It is the default package also know as heart of the java. Like any othe package we don't need to import this package. It is automatically imported. It contains language support classes which defines primitive data types, math operations.

Example:- System, String, Object .. etc.

2. java.util :

This package is used to implement data structure in java. It contains utility classes also known as collection framework. To take user input we need scanner class which is defined in this class.

Example :- Linked list, Stack, vector, etc.

3. java.io : IO stands for input/output

this package is useful for input/output operations on file.

Example :- File, FileWriter, FileReader etc.

4. java.applet :

GUI related & contains classes for creating applets.

6. java.awt :

awt full form abstract window tool kit. Used to develop GUI application.

Example :- frame, Button, Textfeild etc.

7. java.net :

contains classes for networking operations.

Example :- URL, InetAddress, URL Connection etc.

8. javax.swing :

An platform independent and easier to create GUI application. It is extended version of awt.

Example :- JFrame, JButton, JTextfeild etc

9. java.SQL : Database related classes.

Example :- ResultSet, Statement, Connection etc

Example :

keyword to import any packages

import java.util.ArrayList;

import this class from the package

```
public class Main {
    public static void main(String args[]){
        ArrayList<Integer> a =
                new ArrayList<Integer>();
        a.add(1);
        a.add(2);
        a.add(3);
        System.out.println(a);
    }
}
```

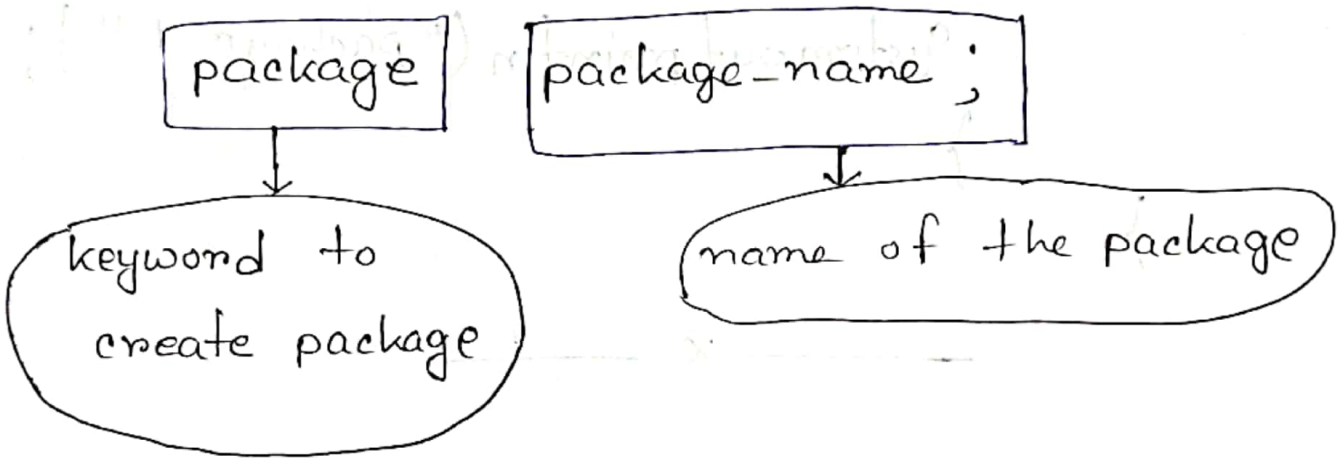
output :

[1, 2, 3]

User-defined package :

The package which is defined by user it's called user-defined package.

Syntex :



member acessibility :

modifier	within class	within package	outside pkg by subclass	outside package
public	✓	✓	✓	✓
default	✓	✓	X	X
protected	✓	✓	✓	X
private	✓	X	X	X

Example :

```
package package1;

public class X {

    public void show() {
        System.out.println("package 1");
    }
}
```

——— x ———

```
package package2;

public class Y {

    public void show() {
        System.out.println("Package 2");
    }
}
```

——— x ———

```
package packageMain;

import package1.X; // import only X class

import package2.*; // import all the class
                   in the package.

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

        X x1 = new X();
        Y y1 = new Y();

        x1.show();
        y1.show();

    }

}
```

output :

```
package 1
package 2
```

We can also create package inside a package.

Let's assume we have "package Main" or named package. To create package inside this package the naming convention will be —

```
package Main.packageSub;
```

Syntex : —

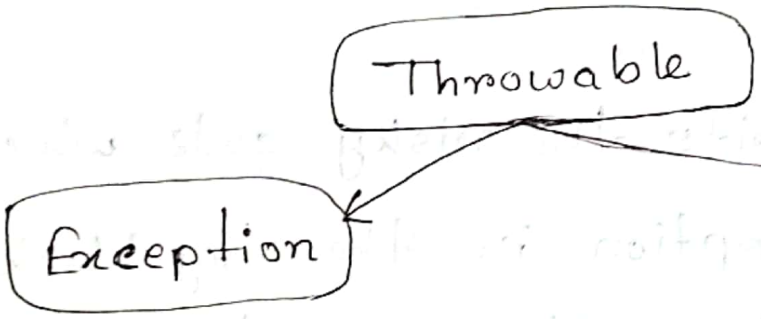
```
package packageMain.packageSub.*;
```

```
package Main
    └ packageSub
        └ ....
```

Exception Handling :

An exception is an unexpected event that occours during program execution. It cause the program to terminate abnormally.

Exception Hierarchy :



-> RuntimeException
-> IOException
-> InterruptedException

-> StackOverflowError
-> OutofMemoryError
-> IOError

** RuntimeException :
-> ArithmeticException
-> NullPointerException
-> NumberformatException
-> IndexOutofBounds.Exception

Exception handling is a mechanism to handle this run time errors.

Java provides some mechanism to work with exception :-

1) try 2) catch 3) throw 4) throws
5) finally

* We have to write the risky code which can cause exception in the try block. try block will throw the exception. try { }
* We have to catch the exception using catch() block.

```
catch (Exception e) {
    //
}
```

object

superclass it can catch any type of exception

* We write finally { } after try & catch finally block will be executed no matter what always.

Example :

```
public class Main {
    public static void main (String [] args){
        try {
④           int divideByZero = 5/0;
            System.out.println ("try block");
        }
        catch (ArithmeticException e) {
            System.out.println ("can't divide by zero");
        }
    }
}
```

// ArithmeticException is a subclass of Exception superclass it catches only Arithmetic type exception.

Here, if a number is divided by zero it throws Arithmetic exception. if we don't handle it will throw error and stop the program in 4th Line.

Multi-catch & finally :

Example :

```
class Test {
    public static void main (String [] args) {
        try {
            int a[] = new int [5];
            a[5] = 30/0;          → Exception
        }
        catch (ArithmeticException e) {
            sout ("task 1"); }
        catch (ArrayIndexOutOfBoundsException e){
            sout ("task 2"); }
        catch (Exception e) {
            sout ("task 3"); }
        finally {
            sout ("finally Block");
        }
    }
}
```

Output :

task 1

finally Block

// only 1st catch Block will be executed when there is multiple catch Block.

Custom / User-defined Exception :

```
class InvalidAgeException extends Exception{
    InvalidAgeException(String s){
        super(s); // parent class constructor
                  will get s
    }
}

public class Test {
    public static void age(int age) throws
            InvalidAgeException {
        if (age <=18) {
        -> throw new InvalidAgeException
                ("Invalid Age!");
        }
        else {
            System.out.println("valid");
        }
}
```

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

    try{
        age (2 ); // static method
    }
    catch(InvalidAgeException e ) {
      System.out.println (e.getmessage());
    }

  }

}
```

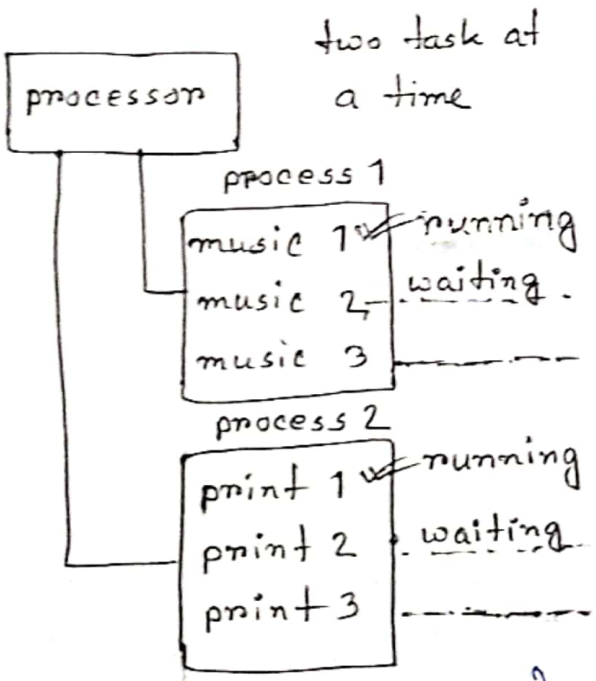
Thread :

A thread is a lightweight subprocess, the smallest unit of processing.

Java provides 'Thread' class to achieve thread programing.

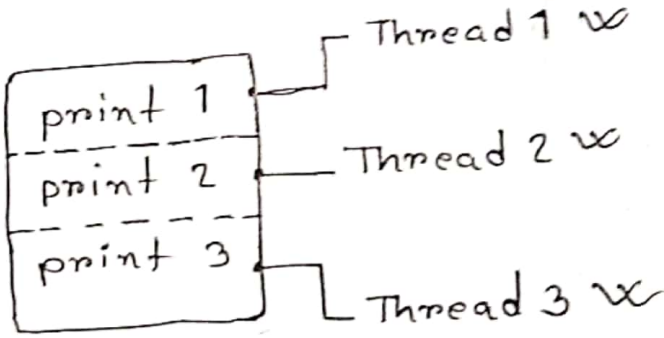
Multitasking

process based



{wait until Print 1 finishe}

Thread based



{3 print will be done at the same time.}

{one process divided into sub process.}

Multithreading :

It is a process to execute multiple threads at the same time. Multithreading is thread based multitasking.

Advantages :

1. It doesn't block the user because threads are independent and it can perform multiple operation at the same time.
2. It saves time
3. It doesn't affect other threads if an exception occours.
4. Unlike multiprocessing, in multithreading threads shared same memory area
5. thread is lightweight.
   for example, I have to listen all music & print all the document. If I do two thing at a time processor will feel more pressure.

There are two ways to create a thread:

1. By exteding Thread class
2. By implementing Runnable interface

Methods in Thread class:

start() - to start the execution of thread
run() - to specify the work we want the thread to perform
sleep() - to pause for a certain time
join() - it waits for a thread to die
currentThread() - returns referance of currently running thread
isAlive() - check if thread is alive
stop() - stop the thread

interrupt(), getPriority(), setPriority(), getName(), setName(), suspend(), resume()

Implementation Using Thread class :

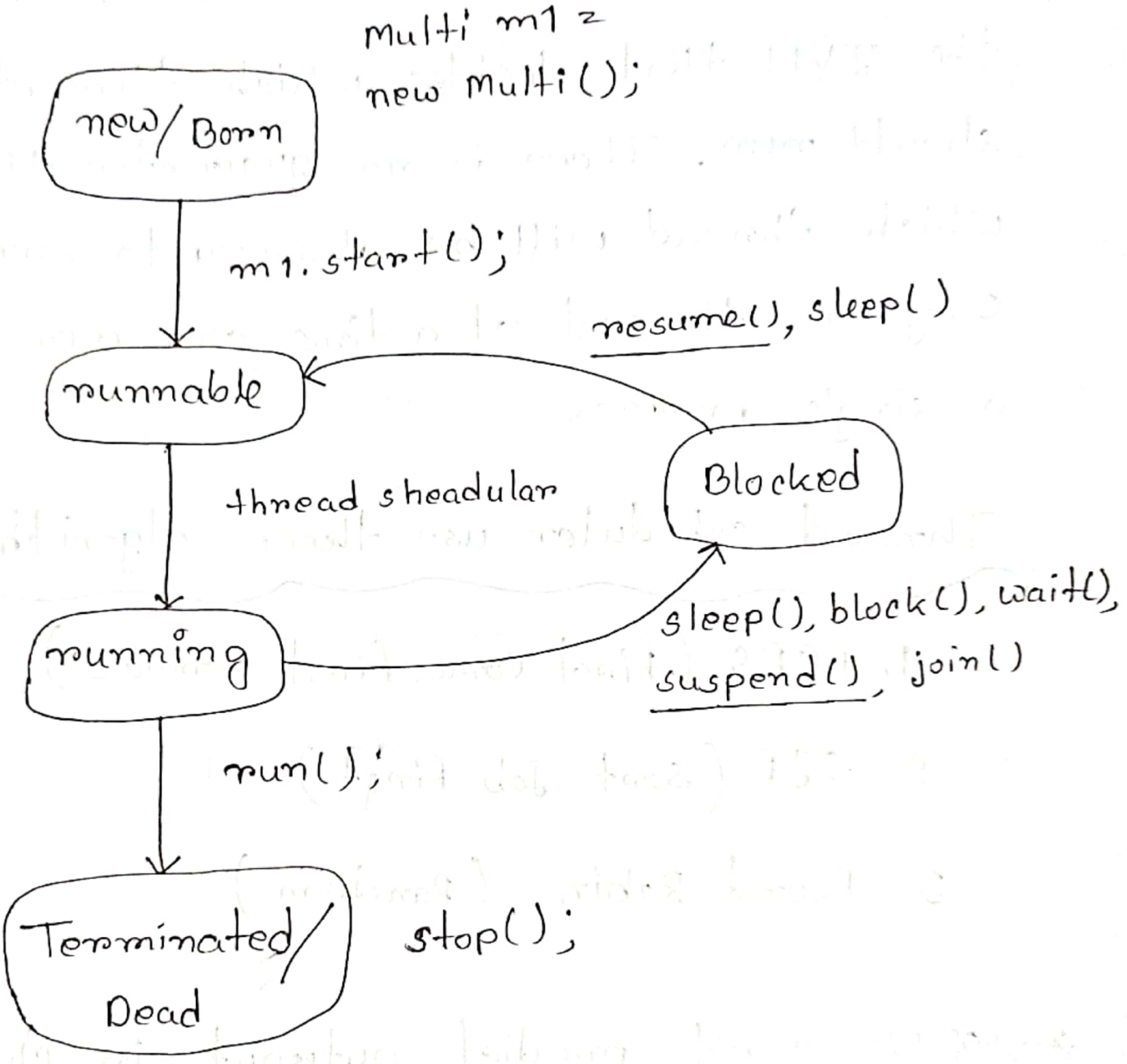
```
class multi extends Thread {

    public void run() {
        SOUt ("Thread running.");
    } // thread define in run method..

public static void main (String [] args) {

    multi m1 = new multi();

    m1.start(); // start thread

    }
}
```

Using Runnable interface :

```
class Multi implements Runnable {
    public void run(){
        sout("thread running");
    }
psvm() {
    Multi m1 = new Multi();
    Thread t1 = new Thread(m1);
    t1.start();
    }
}
```

// Runnable interface have only run() method. To start we need start() from Thread class.

Thread life cycle :



Thread scheduler :

Thread scheduler in java is the part of the JVM that decides which thread should run. There is no guarantee that which thread will be choosen to run. Only one thread at a time can run in a single process.

Thread scheduler use three algorithm :

1. FCFS (first come first service)
2. SJF (Sort Job first)
3. Round Robin (Random)

*** We can't predict output in Multi-threading JVM decides which thread will be executed 1st. Every time we run program output will be different.

Example :

```
class Multi extends Thread {
    public void run(){
        try {
            for (int i=0; i<5; i++) {
                sout(getName());
                Thread.sleep(1000);
            } // executed after every 1sec.
        }
        catch (InterruptedException e) { }
    }
}

public class MultiThread {
    PSVM() throws InterruptedException {
        Multi t1 = new Multi();
        Multi t2 = new Multi();
```

JVM will handle the exception

```
t1.setName("Thread 1");
t2.setName("Thread 2");

t1.start();   t1.join();// t1. thread executed completely other thread will in be waiting period then other thread will be executed in any order
t2.start();

for (int i=0; i<5; i++){
    sout("main Thread");
    Thread.sleep(1000);
    }
  }
}
```

Thread Priority :

We know, java thread shedular choose which thread will be executed.

It is possible to set priority of thread.

By setting priority we can choose which thread we want execute first or any time.

- Thread class provide three predefined final static variable → default value.

  Thread.MIN_PRIORITY → 1

  Thread.NORM_PRIORITY → 5

  Thread.MAX_PRIORITY → 10

  we can set the value from 1 to 10.

Example :

```
psvm ( ) {
    Multi t1 = new Multi();
    Multi t2 = new Multi();
    Multi t3 = new Multi();

    t1.setPriority(2);
    t2.setPriority(5);
    t3.setPriority(Thread.MAX_PRIORITY);

    t1.start();
    t2.start();
    t3.start();
}
```

output :

```
Thread 3 . . .
Thread 2 . . .
Thread 1 . . .
```

Synchronization :

It is mainly used to prevent interference of multiple thread.

There are two types of synchronization :

1. method level synchronization
2. Block level synchronization

The main purpose of synchronization is to overcome the problem of multithreading when multiple threads are trying to access same resource at the same time it may provide wrong result sometime.

Example: (method level synchronization)

```
class Table{
    public synchronized void printTable(int n){
        for(int i=1; i<=5; i++){
            sout(n*i);
        }
    }
}

class MyThread1 extends Thread {
    Table t;
    MyThread(Table t){
        this.t=t;
    }
    public void run(){
        t.printTable(5);
    }
}
```

```
class MyThread2 extends Thread {

    Table t;
    MyThread2(Table t) {
        this.t = t;
    }
    public void run() {
        t.printTable(10);
    }
}
public class Test {
    psvm() {
        Table obj = new Table();
        MyThread1 t1 = new MyThread1(obj);
        MyThread2 t2 = new MyThread2(obj);
        t1.start();
        t2.start();
    }
}
```

output :

5
10
15
20
25
} MyThread1

10
20
30
40
50
} MyThread 2

When 1 thread is executing another waits ~~to~~ until it finishes.
Without synchronization, the output would have been unpredictable.

Example: (Block level synchronization)

```
class Table {
    void printTable(int n) {
        synchronized(this){
            for(int i=1; i<=5; i++) {
                sout(n*i);
            }
        }
    }
}
```

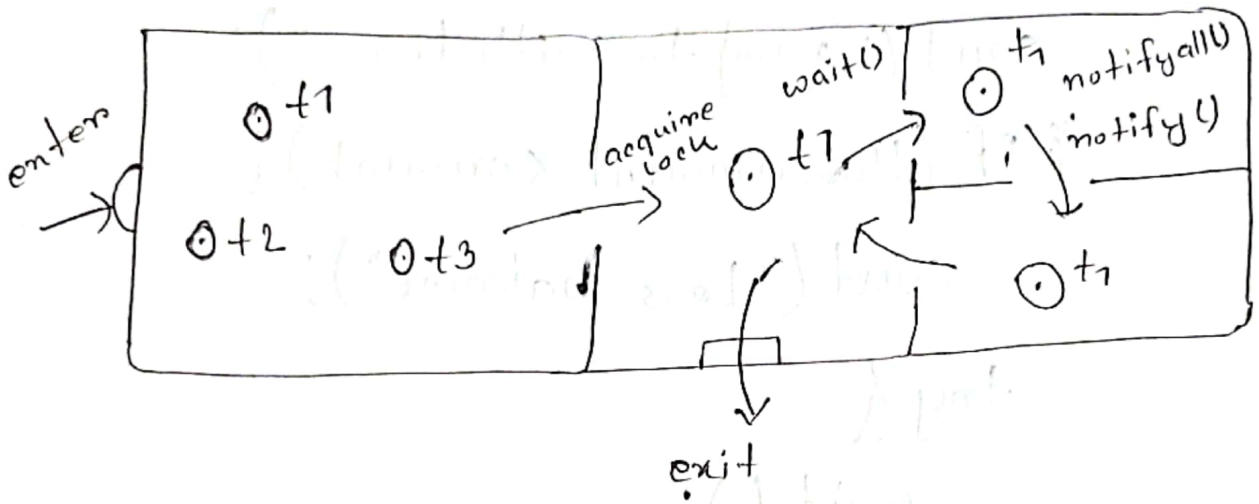
// Here we put few lines we want to synchronized. waiting period is less then method level synchronization.

Example : (static synchronization)

```
class Table {
    synchronized static void printTable(int n) {
        for (int i=1; i<=5; i++) {
            sout(n*i);
        }
    }
}
```

Interthread communication :

Used to allow synchronized threads to communicate with each other.



wait() ; causes current thread to release the lock & wait until its work is done on notif(), notifyAll() is called.

notify() ; wakes up a single thread that is waiting.

notifyAll() ; wakes up all the thread waiting

Example :

```
class Customer {
    int amount = 10000;
    synchronized void withdraw (int amount){
        sout("going to withdraw");
        } if (this.amount <amount){
            sout("Less balance");
        try {
            wait();
        }
        catch (Exception e){
        }
        this.amount -= amount;
        sout ("withdraw done");
    }
```

```
synchronized void deposit (int amount){
        sout (" going to deposite ");
        this.amount += amount;
        sout (" deposit done ");
        notify();
    }
}
class Test {
    psvm (){
        final Customer c = new Customer();
        new Thread (){
            public void run (){
                c.withdraw (15000);
            }.start ();
        new Thread (){
            public void run(){
                c.deposit (10000); }.start();
    }
}
```