Object Oriented Systems Design

⇒ **Machine Language Programming :** 1000100010 0

⇒ **Assembly Language Programming :**

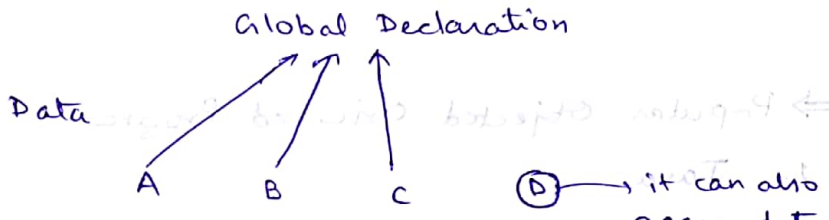
Assembly LaP was better than machine LaP in a sense that assembly LaP were more like English.

EX :

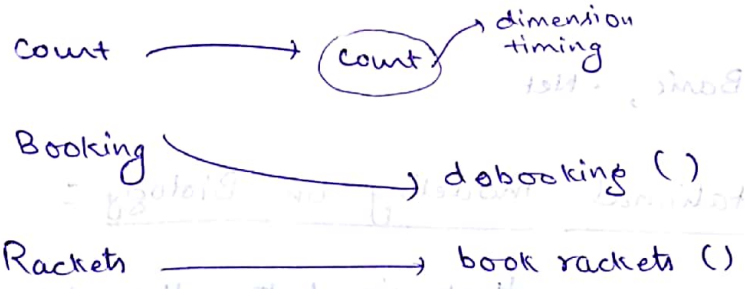
```
ADD  A, B
Move B, C
Jump 9000
```

⇒ **Structured / Procedural Programming :**

C Code :

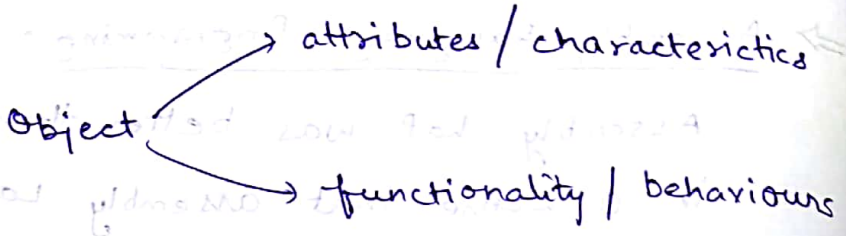


Ex :



⇒ **Object:** An object is an entity with data and code.

Typically, objects encapsulates some or all of the data and functionality.



EX: Design and build a computer Hockey Game.

Soln: Object: Players, Ground.

Characteristics / Attributes: Position, Skills, Speed, Height, weight, no. of goals.

Functionality / Responsibility: Pass the puck, Shoot, Skate forward, Skate backward

⇒ Popular Objected Oriented Program:

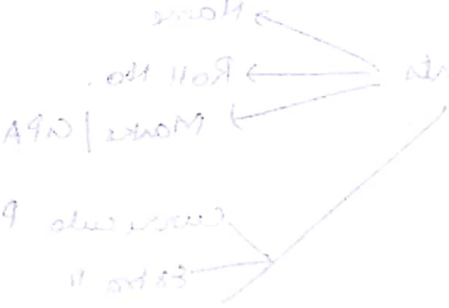
1. Java
2. Python
3. C++
4. Visual Basic, .Net

EX: Computational modeling on Biology:

→ Write a program that simulates the growth of virus population in humans over time. Each virus cell reproduces itself at some time interval. Patients undergo some drug treatment to inhibit the reproduction process and clear the virus from the body. However, some of the cells are resistant to drug and may survive.

Soln :

Objects	Characteristics
Patients	infected with virus population.
	immunity to virus (%)
	Functionalities / Responsi-bilities
	take drug
	diet
Virus	reproduces itself
	resistance (%)
	functionalities
	reproduce
	survive.

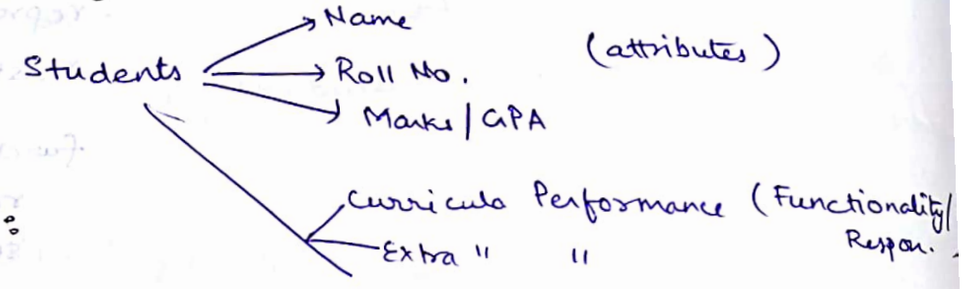


In programming, we map attributes / characteristics to nothing but variables or let's say data members and map functionality with functions.

2. Class : A class is a construct that is used to describe code that is common to all objects of that class. It is the specification of the object.

EX:

⇒ Student Management System :



Student S :

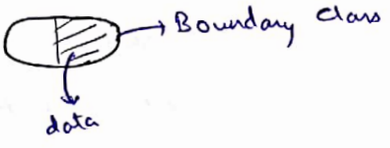
That means all these attributes and behaviour are related inside that class. Object is an instance of type student class.

3. Abstraction :

Abstraction has two view points. The First One is to take the necessary details and ignoring the rest. The second view point is to create objects and class and use them without even knowing what's the code behi That is an another aspect of Abstraction.

4. Encapsulation :

(same as example of capsule (protecting, privacy))



This protection of data is called Encapsulation.

class Student

data private
- name of the student
- Roll No.
- Marks

public (can't be → if private, then it is of no use)
- CurriculaPerf ( )
- Extra " " ( )
- change Marks ( )

→ Boundary class

These public functions are called as Interface.

for eg:- In AC, company is giving you controlled access and that is just by remote. So, in this case remote is the 'interface'.

Encapsulation
- → data & functionality bound togethers in a class
- → Controlling the access over data → Interface

5. Inheritance :

class Health
Object (Father)

Object (child) → properties (data)

↳ accesses all the data from father

reusibility of the code i.e. it can access all the function, data members and code.

EX :

```
class Vehicle {

    Private / Protected :
        String License / No. Plate ;
        int Year

    Public :
        Vehicle (const. string & mylicense, Const int myyear)
        { return license + "from" + stringy(year); }
        { return Make + " ", string }.
}
```

EX :

```
class Car
{
    Public : Vehicle
        String style ;

    Public : Car ( Const String & mylicense , Const. int year
                   const. string & mystyle )
        : vehicle (mylicense, my Year), Style (mystyle){

        Const string & getstyle ( )
        { return style, }

};
```

6. Polymorphism :

a greek term "many forms / many shapes".

Polymorphism is used to facilitate the use of the software. Via polymorphism same function can be applied to different objects to give different results.

(eg:- Vegetable SuperMarket, washing)
(all attacks in different way with only one function)

⇒ Softwares (S/w) are large and complex and they are developed through four phases.

1. Planning
2. Analysis
3. Design
4. Implementation

⇒ The System development life cycle (SDLC) is the process of understanding how an Information System (IS) support needs, build up and delivered to the users.

The terms used for them are :

1. Problem space distribution.
2. Programming language abstraction
3. Algorithm abstraction.
4. Procedural language / OOP approach.
5. Data abstraction
6. Software abstraction.

⇒ The three fundamental principles by which a software can be developed are :

1. Planning / design
2. Abstraction, i.e. concepts of hiding the information with the recognition of structures.
3. Encapsulation / Data privacy

⇒ Programming Paradigms :

a)- Machine language ; 101001000100

b)- Assembly language.

c)- Structured / Procedural language

d)- Object Oriented Programming Language

⇒ In OOP, all the objects are independent, they are connected to one another through the messengers. It's not possible in our usual procedural/structured programming.

```
class Student
{
    roll no.
    name
    marks

    Curr. Perf. ( )
    Extra Pef ( )
}
```

⇒ Limitations of Structured / Procedural Lang :

1. Does not effectively addresses data abstraction and information hiding.

2. Does not response to changes in the program or problem domain.

3. All data used by them end up being global to entire software. This means any change in their representation tends to affect all ~~our~~ other functional programming.

⇒ Softwares quality measurement criteria :

1. Correctness : Program should meet their specifications correctly.
2. Reilience and Reliability (Robust) : It should not fail quite often.
3. Maintainability : Program should be easy to maintain and extended as required.
4. Interportability : Program should be readily compatible with other systems/platforms in fact they should be open system.

(By open system we mean that what we have built or others have built and whatever way we use it, we should expect it to work in every system with minimum effort required).

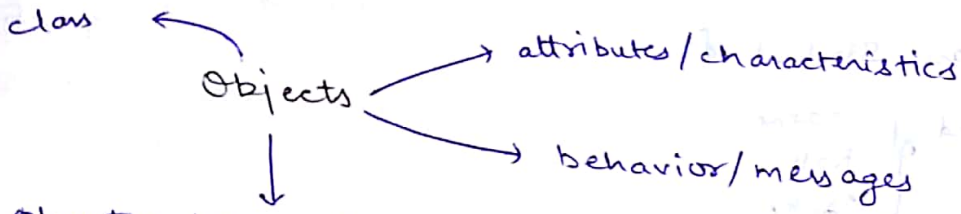
5. Reusability and Generality
6. Efficiency
7. Verifiability
8. Security :
   Data Knowledge and even functions may require selective and effective concealment.
9. Integrity :
   System and program need protection against inconsistent updates or malwares.
10. Friendliness :
    Program should be easy to use by a majority of users.
11. Describability :
    It should be easily documented.
12. Understandability :
    It should be easy to understand.
    Or follow without any technical details.

⇒ Events which influenced the OOP development :

1. Data privacy
2. Advances in computer architecture, compilers and operating systems.
3. Advances in programming languages such as SIMULA, ADA, C++, JAVA, objective C and so on.
4. Advances in programming methods,
   i.e from machine lang. → Assem. lang → Struc. lang. → OOP

⇒ Benefits of OOP Approach :

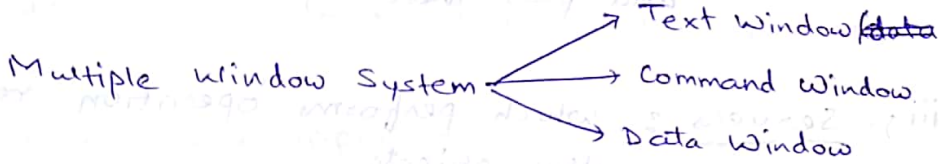
1. Reduces the total (life cycle) software cost by increasing the programmer's productivity and minimizing the maintenance cost.
2. Softwares resist both accidental and malacious attempts to their corruption or, failure.
3. Modifications and extensions are performed in a much easier way as the object are self-entity.
4. Enhances understandability and maintainability as objects and related operations are localised.
5. Provides a mechanism for formalizing the model of reality i.e, it makes direct and natural correspondence between real world and its models.
6. The data centric design approach and proceses to secure privacy.
7. Abstracting only the necessary information and decreasing the problem scalability.



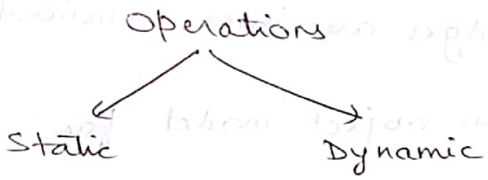
⇒ OOP software development step:

1. Identify the objects and their attributes by recognizing major actors, agents and servers in the model of reality. There can be several objects of similar type, they are represented by a class of objects

EX:



2. Identify the behavior of the objects. This implies the methods through which the objects perform every operation required for that object.



⇒ Static operations: Operations that are performed and required from other objects.

Ex: open/close, text field.

⇒ Dynamic Operations: Changes in the objects under constraints of the application in hand.

EX: Time

3. Establish the visibility of each object to other objects. All objects must have internal and external visibility.

EX:

class Student {

private/protected { name, Roll, Marks

Curri.Perf ( ), ExtraCurrPerf ( ) } : public

}

class NCR

(i)- Actors : It sends messages telling other objects what operations they are to be perform.

(ii)- Agents : which accepts operations from one object and pass these operations to other objects.

(iii)- Servers : which performs operation requests by other objects.

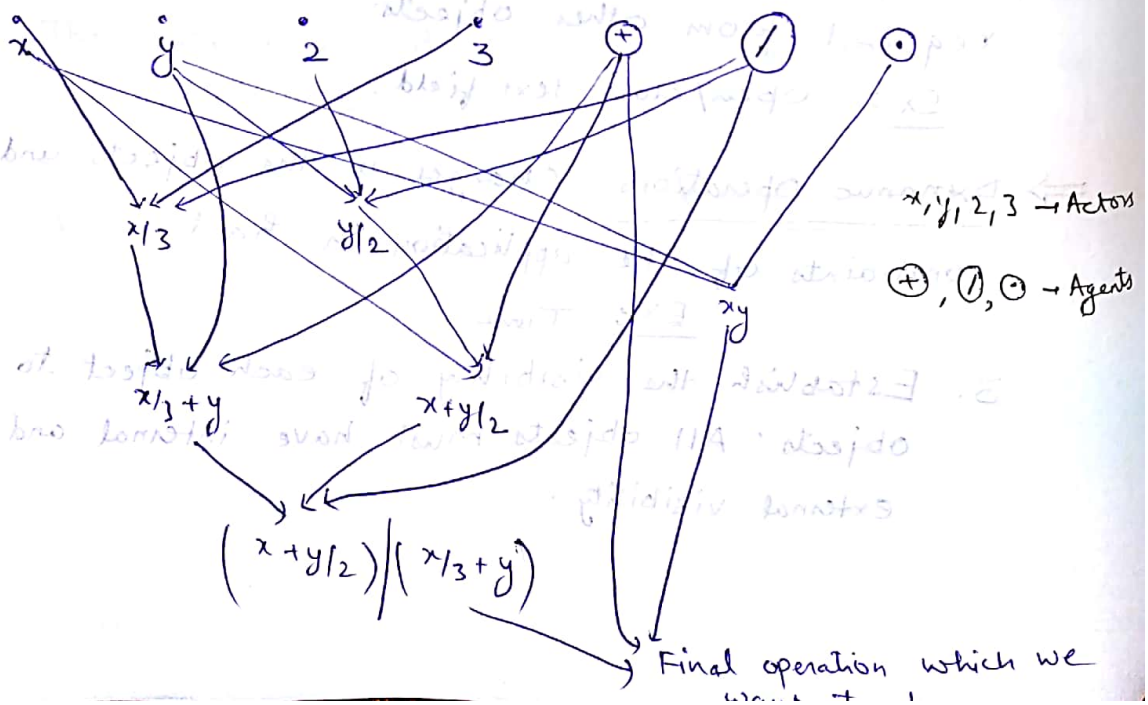
1/8/18

⇒ OOP Modeling :

1. Object Model : It is a graph where nodes are objects and the edges are inter relationships between them.

EX: Draw an object model for

$$M = (x + y/2) / (x/3 + y) + xy$$



2. Dynamic Model :

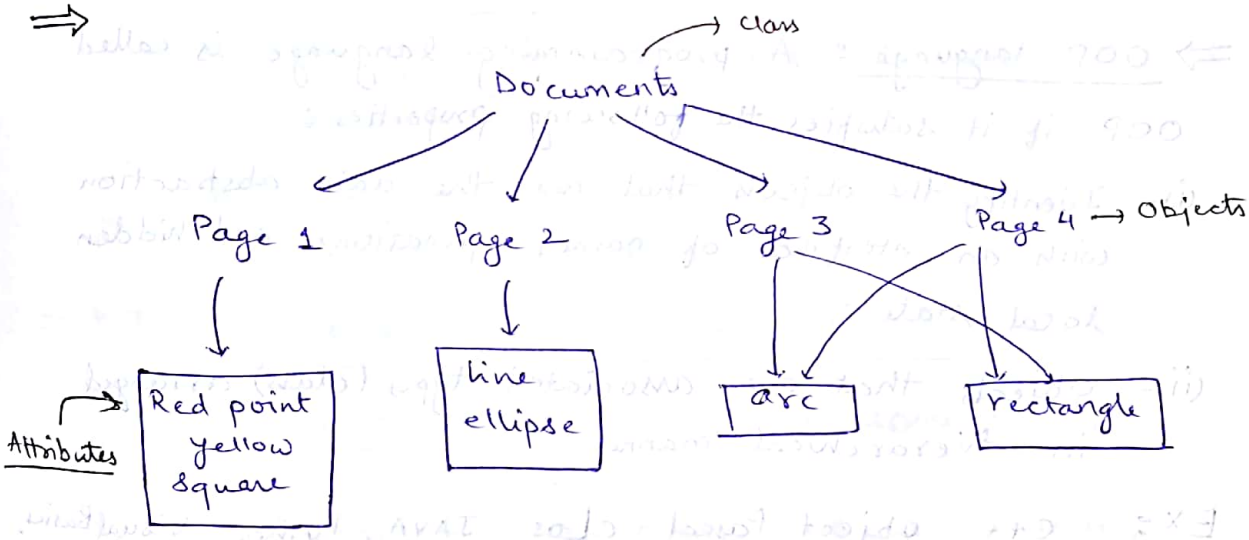
It is also a graph where nodes are states and edges are transitions which occur due to occurence of various events.

3. Functional Modeling :

It is a data flow graph (DFG) indicating operators and operands and the order in which they have to be executed.

EX 2: Draw an object model for a document giving following informations : It has 4 pages. The first page has a red point and yellow square, the 2nd page contains a line and an ellipse, and, an arc and a rectangle appear on the last two pages displayed on it.

⇒



⇒ **Issues in OOP :**

(i)- What exactly are the classes and objects involved?

(ii)- How classes and objects are relevant to a particular application are identified?

(iii)- How a suitable representation for expressing OOP design can be identified? → (Object Modeling Technique)

(iv)- What processes can lead us to well structured OOP System/design?

⇒ **Developing a OOP S/W :**

To build a language independent design. To think abstractly first and not from Computer point of view. The objective is to analyze problem requirements, design a solution to the problem and then implementation in a programming language.

⇒ **OOP language :** A programming language is called OOP if it satisfies the following properties :

(i)- Identify the objects that are the data abstraction with an interface of named operations and hidden local state.

(ii)- Objects that are associated types (class) arranged in hierarchical manner.

EX : C++, object Pascal, CLOS, JAVA, Python, VisualBasic, .Net

⇒ Properties of an object :

(i)- Self - entity

(ii)- Behavior is characterized by the operations that it performs and required other objects to perform. An operation / a message is a method having a number of parameters.

(iii)- State : Attributes and their values.

EX: class Student {

State { name, roll, marks

(iv)- It's an instance of a class. The class is the form taken by all the objects of similar types classes can have subclasses and all the subclasses are related to by inheritance properties.

(v)- Names : (a)- Several names : called aliases and can be given to one object (at runtime mapped to one name).

b)- one name can be given to several objects (conflicts are resolved at run time).

⇒ Type of operators carried out in OOP :

(i)- constructors : class names with same or number of parameters.

(ii)- Destructor : class names without parameters and are distinguished by ~ operator.

(iii)- Selector : An operator that evaluates the current state of the objects.

(iv)- Iterator : Visit all parts of the objects and return objects.

Object has a restricted visibility

↓

internal view : describes the implementation details.

External view : Behavior

Ex:

1. Dogs : attributes: name, breed, colour, hungry.

   Behavior: barks, fetch thing, wagging.

2. Radios : attributes : ON/OFF, Volume, station.

   behavior : change volume, change station.

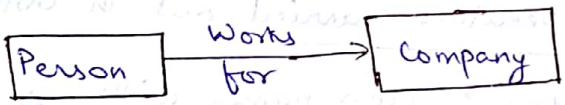
6/9/18

⇒ Object Oriented modelling design notation and concepts :

→ Class & Objects : They are linked together (one to one, many to one, many to many)

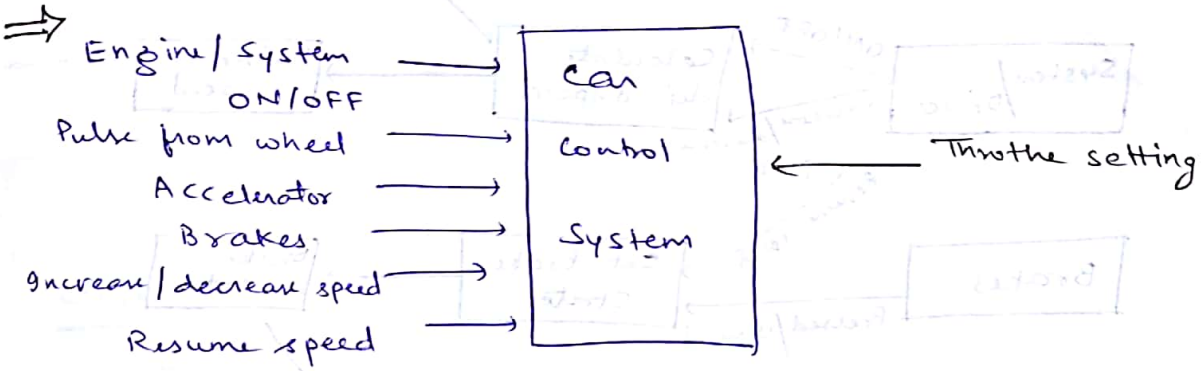
Bullets are utilized to show the multiplicity.

→ Association : An association specifies a group of links between instances of the class. It is inherently bidirectional.



Q. Consider a car control system / model which is used to maintain the speed of a car. Draw the flow chart / block diagram and object model.

⇒



1. System ON/OFF : The program detects if the system is ON or OFF.

2. Engine ON/OFF : The system will only be activated if the Engine is ON.

3. Pulse from the wheel : A pulse is sent for every revolution of the wheel.

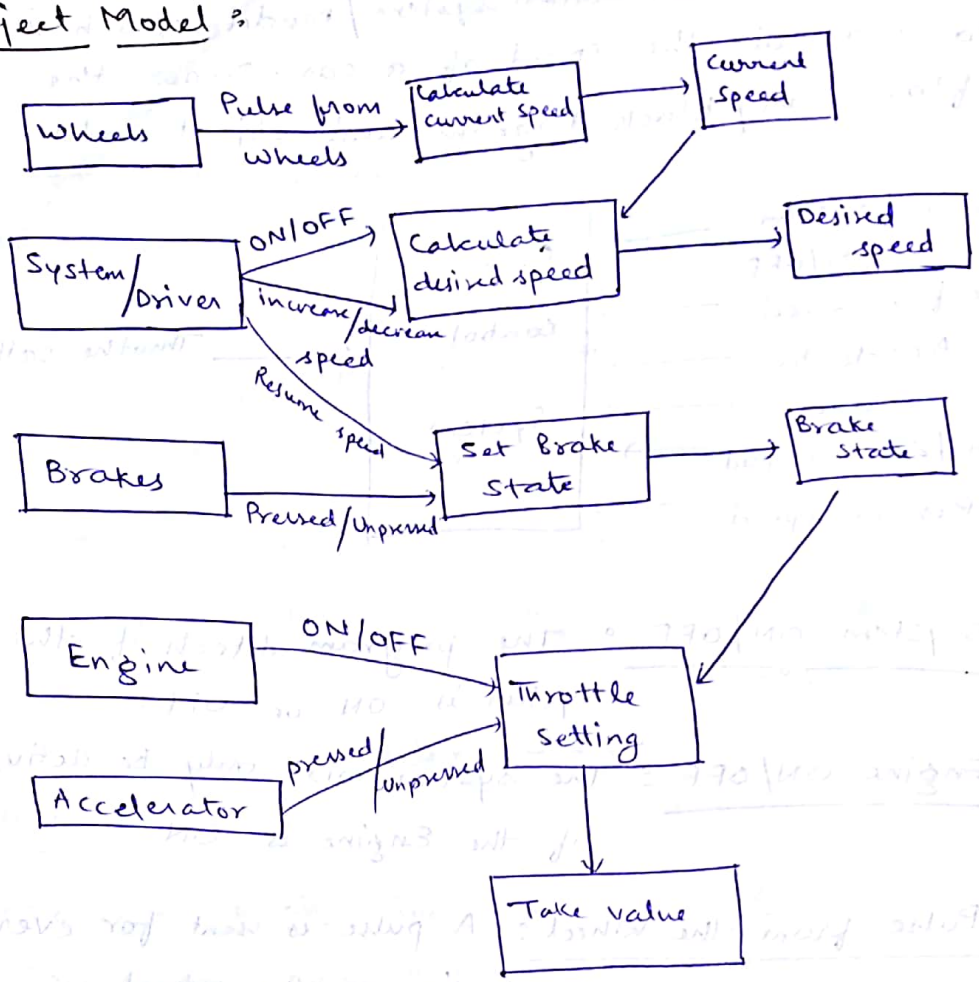
4. Accelerator : It indicates how for / how hard the accelerator is pressed.

5. Brake : if pressed then your program for control system should be able to detect and maintain the speed accordingly.

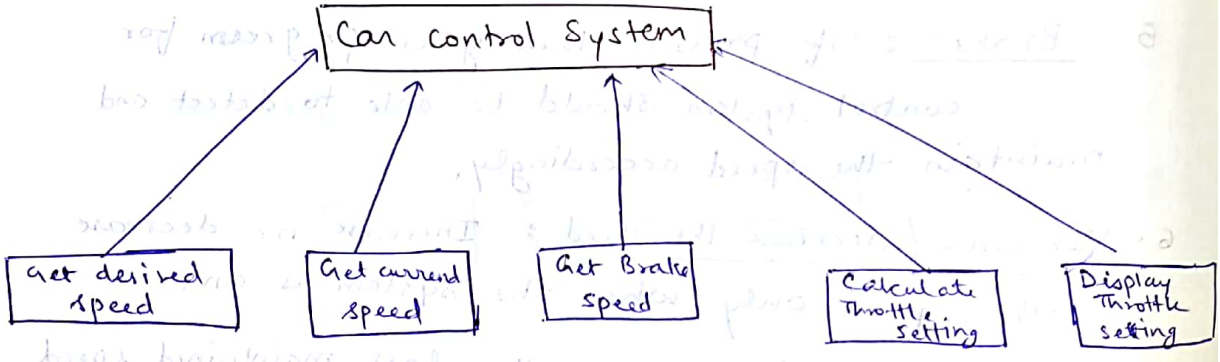
6. Increase / decrease the speed : Increase or decrease the speed only when the system is on.

7. Resume Speed : Resume the last maintained speed.

→ Object Model :



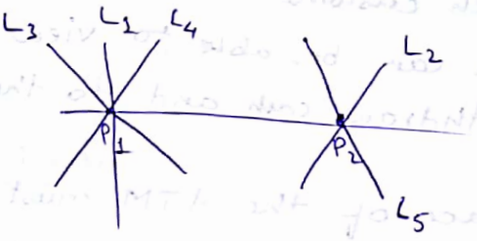
⇒ Functional Model :



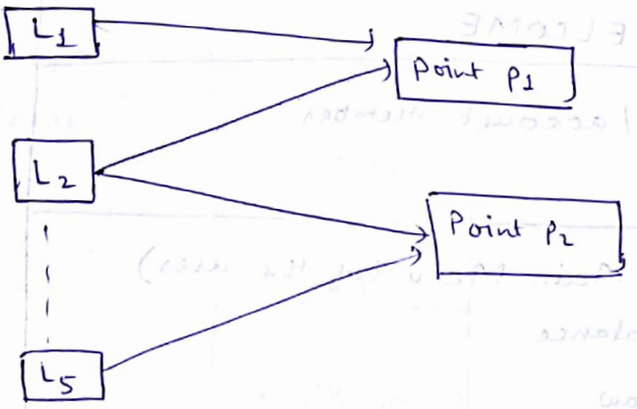
EX: (Many to Many)

Given the intersect points P_1 and P_2.
Find all the lines passing through them.

Soln:



Object Model:



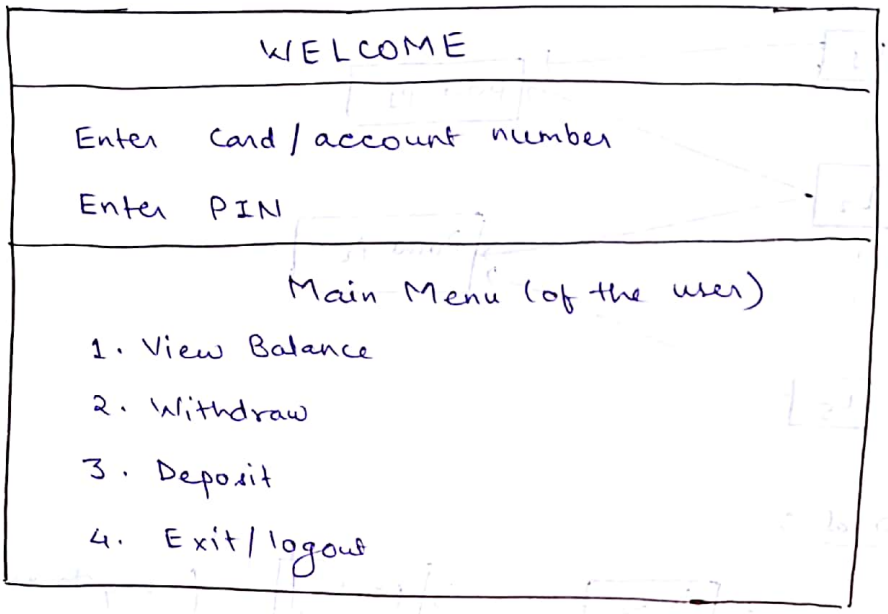
Functional:

(many to many relation)

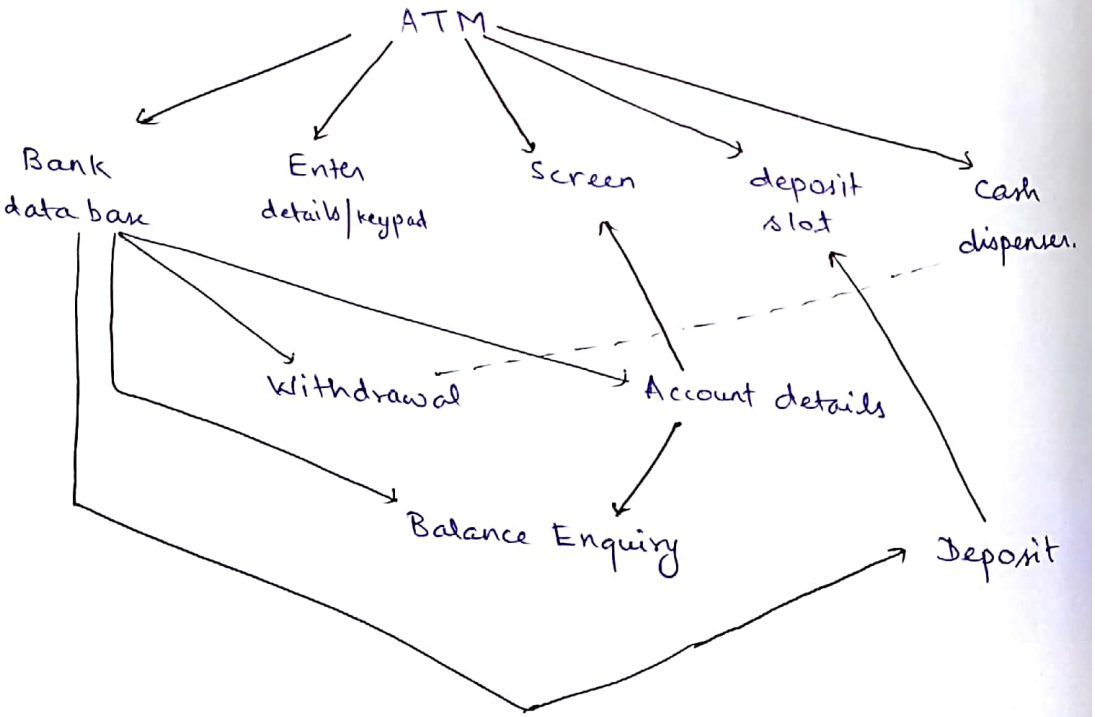
- Question:

Design a model for a local bank which intends to install a new ATM to allow customers to perform basic operations. Each customer can have only one account. A customer can be able to view his/her account balance, withdraw cash and do the deposit.

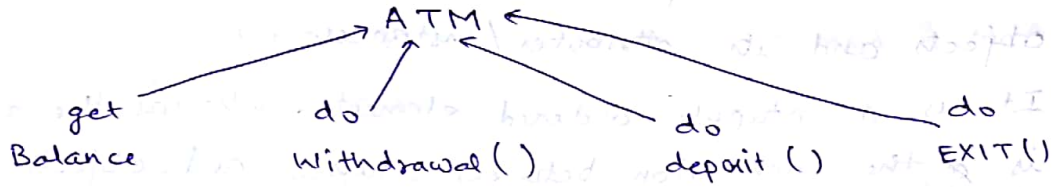
Soln: A user interface of the ATM must contain



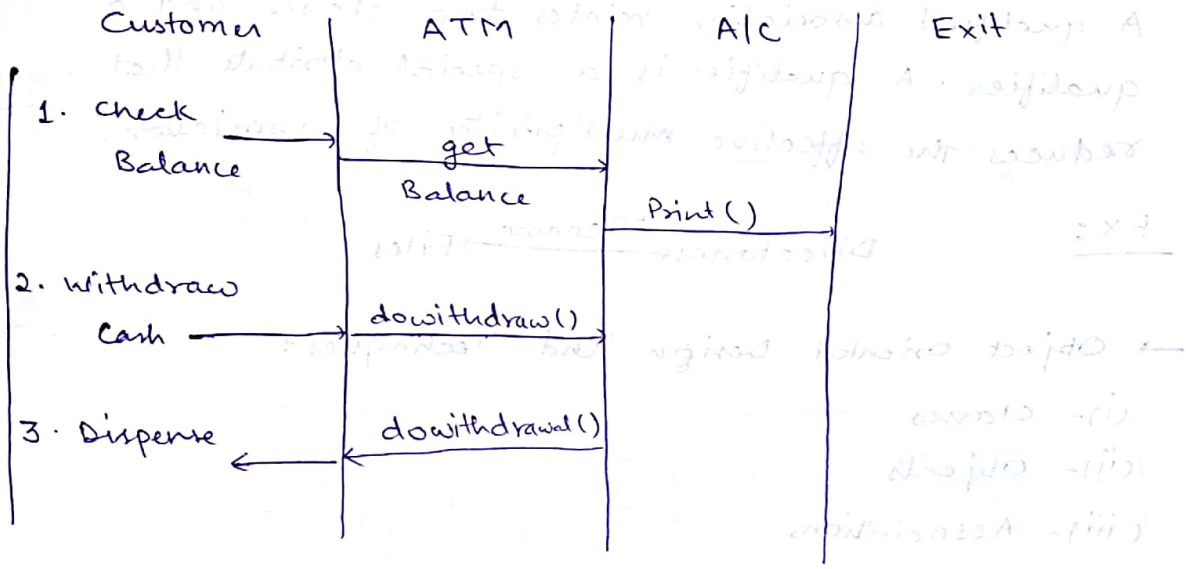
→ Object Model



→ Functional Model :



→ Dynamic Model :



⇒ Object and Classes :

Object is an self entity which stores informations.
Objects are instances of the class.
They can be represented by name, place..... which are treated as a data variable in a class.

EX :

class Person → Rony, 23, Mathe, Delhi.

name
Age
job
Address
get Age ()
get Address ()

Each object will/may have different attributes

⇒ **Links and Association :**

A link is a conceptual or a physical connection between objects and its attributes / instances.

It is a ntupule ordered elements, whereas the association is the connection between classes and objects.

It can be one-one, many-one, many-many.

⇒ **Qualification :**

A qualified association relates two classes and a qualifier. A qualifier is a special attribute that reduces the effective multiplicity of association.

EX: Directories ←Filename→ Files

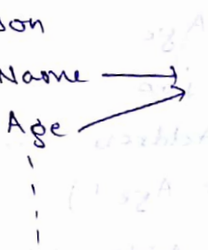
→ Object oriented Design and Techniques :

(i)- classes
(ii)- Objects
(iii)- Associations
(iv)- link attributes
(v)- Ordering —— a)- aggregation b)- Generalization
c)- Inheritance d)- Specialization

(vi)- Qualification

⇒ **Aggregation :** It is a part of a relationship. This means the components of something is termed as a part of the assembly representing the entire component.

EX:



A name, an argument list, part of a program, compound statements / functions.

⇒ Properties of Aggregation :

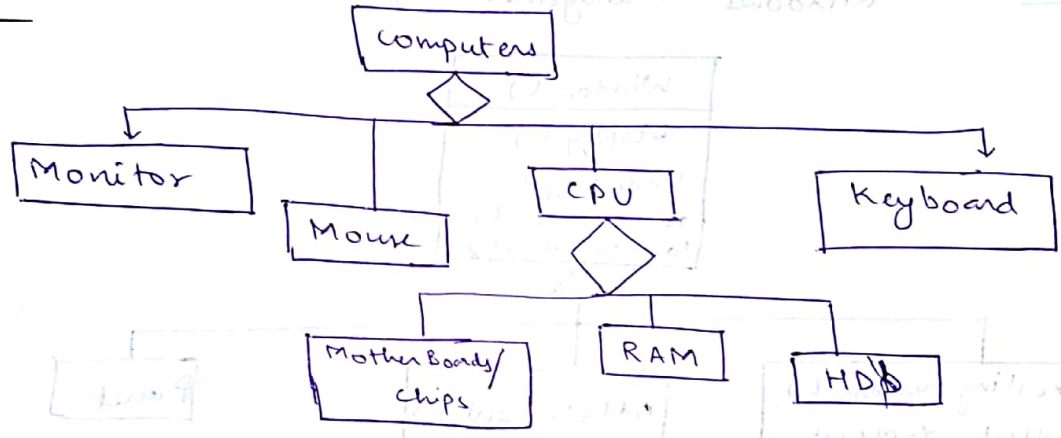
(1). Transitive : If A is a part of B and B is a part of C, then A is a part of C.

(2). Anti-symmetric : If A is a part of B, then B is not a part of A.

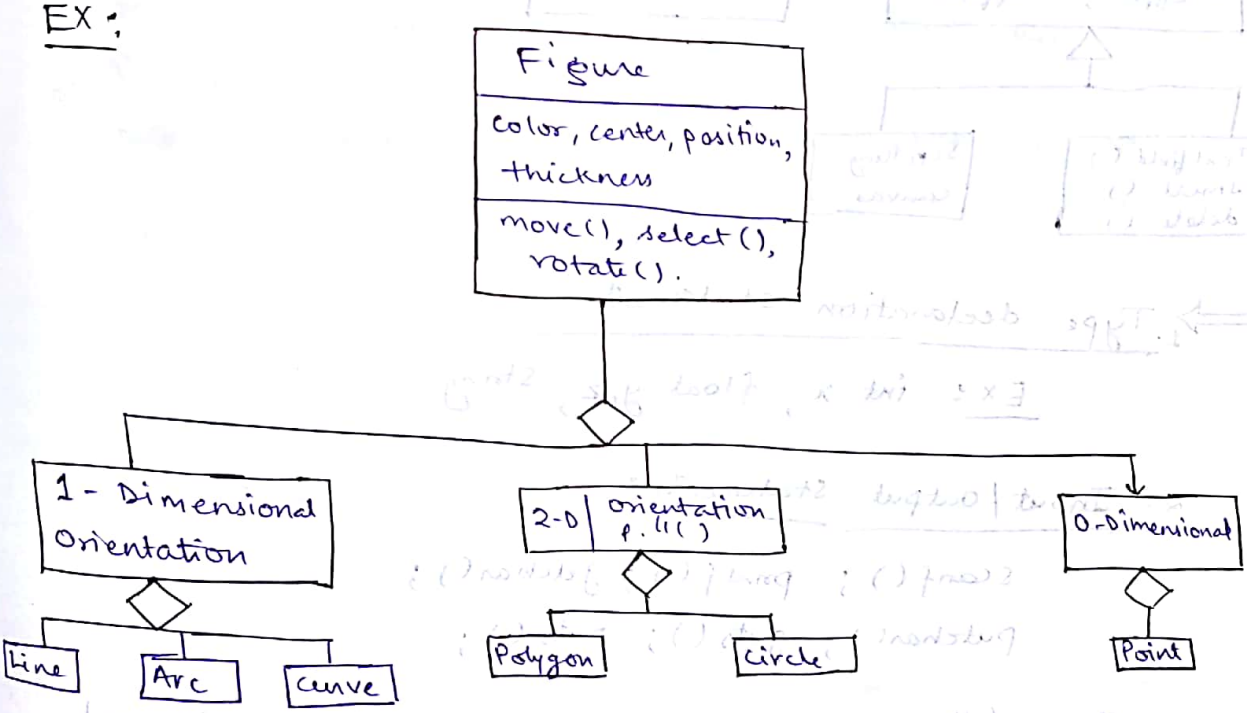
To represent aggregation notation, we use ◇ (It also represent the end of assembly).

EX 1 :

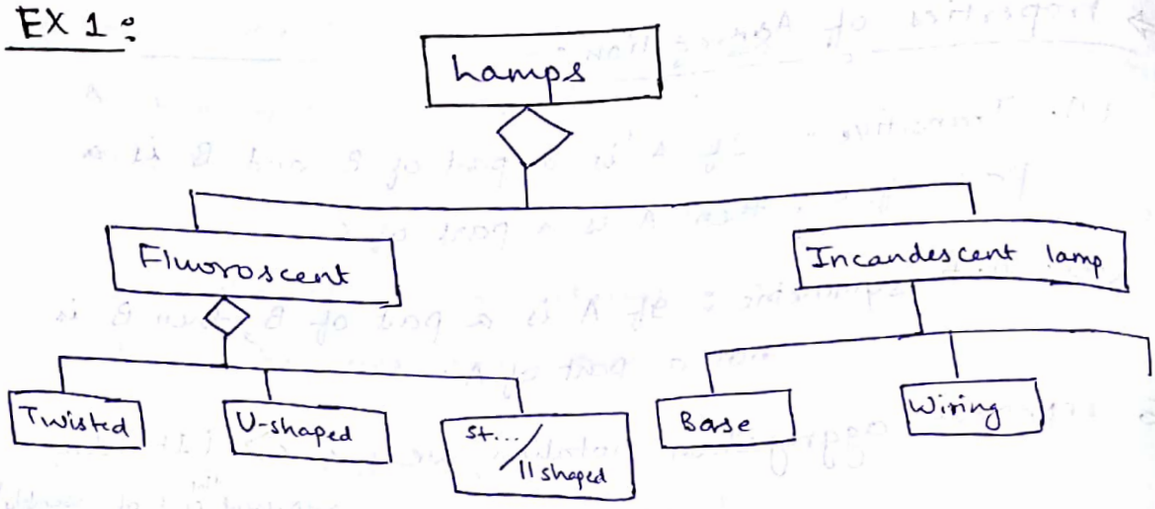
EX :



EX :



EX 1 :

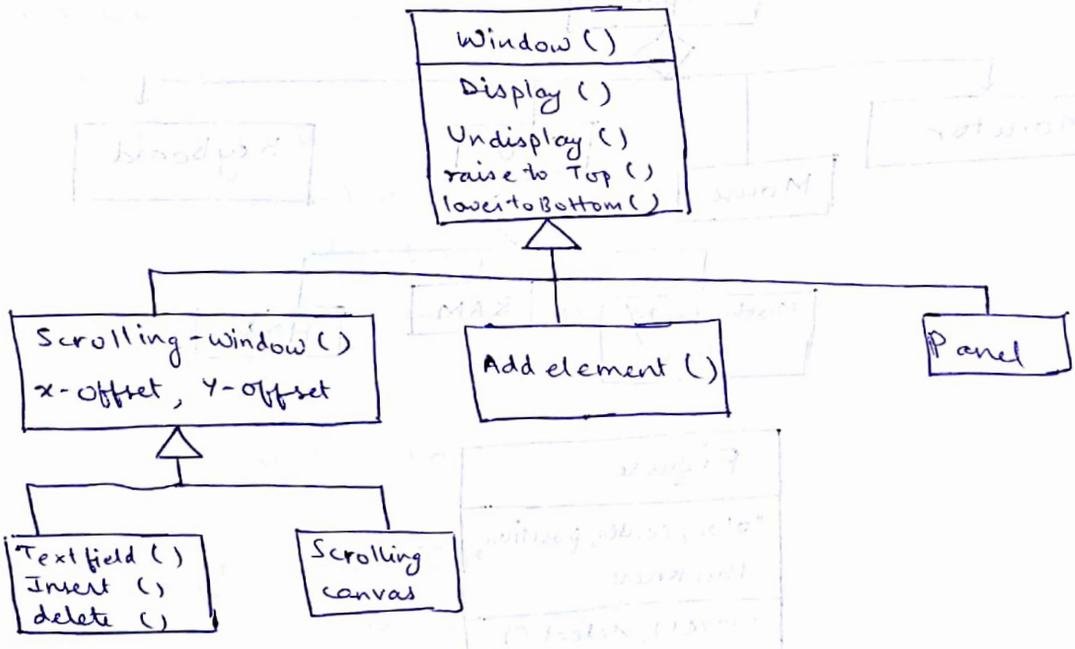


13/08/19

⟹ Aggregation :

Part of relationship, transitive and anti-symmetric.

EX : Windows Management System



⟹ 1. Type declaration statements :

EX : int x, float y, z, string

2. Input / output statements :

scanf() ; printf() ; getchar() ;

putchar() ; gets() ; puts() ;

Scanf ("<format string>", list of variables with and preceeding them).

1. formatted

2. Unformatted : Data means an uniform stream of bytes. Meaning these datas are interpreted by the programs which access them. This could be a binary data, text data, command data or even a program.

[EX : {read() / write(), scanf(), printf(), getchar(), putchar()...} .

EX: "Hello" → array of characters

```
char a[6];
a[0]="H"; a[1]="e"; a[2]="l"; a[3]="l"; a[4]="o";
a[5]="\n".
```

EX:

```
char c, d;
getchar(c);
putchar(d);
```

scanf() =

scanf ("<formatted string>", list of variables),

printf ("<formatted string>", list of variables),

printf():

gets() = when a white space character appears on the input line, the string is terminated.

⇒ Libraries :

#include <stdio.h> → for both formatted and unformatted input-output operations.

#include <header.h> → means user defined files.

#include <iostream.h> → it has all operations with input-output operations for object-oriented programming.

Cin → To read from Keyboard.
cout → to display on monitor.
cerr → Errors to be displayed on monitor.
clog → For logging onto hardware.

Ex:

```
cin >> x ;
cout << "Hello" ;
```

"Hello" is considered as a string object and it will display on your monitor.

Extraction operations
Overloading operators

⇒ Manupulators :

(1) endl ; → indicates the end of a line.

Ex:

```
cout << x << endl ;
count << x << y << endl ;
```

(11). setw() → It is to set width.

eg:-

```
int x ;
count << setw(x) << endl ;
```

6

```
int x ;        x = 1234567
count << setw(4) << endl ;
```

Then x will display 1234.

for setw(7) Then x will display 12....7.

for setw(10) then x will display 000 12....7.

(iii)- left/right :

eg: count << left << setw(7) << x << endl ;

x will display = 123...7

for setw(11) x will display = 123...7 0000

for count << right << setw(11) << x << endl

x will display = 0000123...7

20/8/18

⇒ Arithmatical and logical statements :

constants, variables, array names and function references can be joined by some operator's to form an expression.

Operators : * , / , % , + , - . . . .

Logical Operators : != , < , > , <= , >= , && , || , == . . .

eg:- 5.0/2 = 2.5

→ Type casting / co-ercion of data :

Data (type) in an expression.

eg:- (int (i+f))/4 , here i and f are integers

⇒ Control instruction :

(i)- Decision Control statement : if else statement.

if (expression) ⟶ logical statement

Statement 1 ,

else

Statement 2.

if (condition)

S_1

else

S_2.

→ Nested if-else statements:

```
if C1
    if (C2 S2 else S3)
else
    if (C3 S4 else S5)
```

→ Switch Statement:

```
Switch (integer expression)
{
    Case 1 : do this;
    Case 2 : do this;
    .
    .
    .
    default : do this;
}
```

→ Loop Statements:

(i)- while

```
while (expression)  do
{
    .
    .
}
```

expression → conditional

(ii)- for (exp1 ; exp2 ; exp3)

exp1 → initial value

exp2 → conditional value

exp3 → increment value

Statement.

(iii)-

```
do
{
    .
    .
}
while (conditional expression)
```

(iv)- the comma Statement —

```
for (j=0, alpha=100 ; j<50 ; j++, alpha++)
```

j=0, alpha=100 → comma Statement

(v) - Continue Statement :

The control goes back to the first statements of the "for" loop, after getting the statement "continue".

(vi) - Break :

This statement breaks the loop.

⇒ Functions :

```
data type
function name (list of arguments)
{ . . . .
   return ();  ⟶ this will return the value
                  of the function.
}
```

eg :

```
#include <iostream.h>
main()
{
   int a, b, c, d;
   cin >> a >> b >> c;
   d = max(a, b);
   cout << "maximum = " << max(c, d) << endl;
   return ();
}
```

Eg: Write a program to find the factorial of a given number.

→

```
#include <iostream.h>
main()
{
    int a, f;
    cin >> a;
    f = fact(a);
    cout << "factorial = " << f << endl;
}

int fact(int x)
{
    int x;
    int p = 1, i;
    for (i=1; i<=x; i++)
        p = p * i;
    return(p);
}
```

⇒ Arrays :

Array is declared as a storage class.

storage class data type

(i)- Automatic (auto)

(ii)- external (extern.)

array-name [expression] = {val1, val2, ..., valn}

indices

eg:- int a[10] = {1, 2, ..., 10}

so, a[0] = 1, a[1] = 2 ..... , a[9] = 10

eg:- char a[3] : "RED"

a[0] = "R", a[1] = "E", a[2] = "D", a[3] = "\n".

eg:-

```
float average();
{  avg = average(n, list)
float list[100]
float average(int, float());
average(a,x);
int a;
float x();
}
```

n → size of the array

list → name of the array

float average(int, float()); → function declaration

27/8/18

⇒ Operators: >>>, <<<

eg:-

```
input java.io.*
Public class test
{
public static void main (strings args[])
int count = 10;
System.out.println ("count is", +count)
count = count >>> 1;
System.out.println ("count after is", +coun)
```

Java API = Java application programming interface.

Static : implies the class can be accessed without creating the object of the class.

Void : implies return nothing.

print statement : print(); println(); printf() are used to print any statement or a value.

⇒ Multi-dimensional arrays :

storage class data type array name [int exp]
= {val 1, val 2, ..., val n}.

eg:-

int val[3][4] = {1, 2, ..., 12}

val[0][0] = 1
val[0][1] = 2
:
:
val[2][3] = 12

⇒ Arrays can be passed into a function.

```
main()
{ int n, float list[100];
  float average (int, float[]);
  avg = average (n list);
}

float average (a,x)
{ int a;
  float x[];

  return;
}
```

⇒ Pointers :

They are address and variables that hold them are known as pointer variable data type.

* point variable.

eg:-

```
int *a;
float *x, *y;
int count 8
    int * pivar,
    pivar = &count.
```

⇒ Properties of pointer variables:

(1)- A pointer can only be initialized to 0, which is equivalent to NULL defined in stdio.h.

(2)- They are used for creating dynamic data structure.

(3)- pointers can also be used to pass variables to functionals.

(4)- pointers are closely related to arrays, hence array manipulations are simplified if pointers are utilized.

⇒ Operations On Pointers:

1)- One pointer can be arranged to another pointer if they refer to the same data type.

2)- They can be composed by equality (==), relational operators. But pointer values converted/modified to the changed value likewise you can perform (++) or (--) or add integer to a pointer.

eg:- p = 2000
p+4 means p = 2000 + 4·4 = 2016

eg:- (passing pointers into function):
write a program to convert inches into cm by using pointers.

Soln:

```
#include <iostream.h>
void main()
{
    void intocent();          → This is a function declaration without any argument
    double val = 10.0;
    intocent(&val);
}
```

```
void intocnt (double *x)
{
    [redacted] *x = *x * 2.54;
}
int *p;
```

We can say that p is a pointer to the integer array whose subscripts are implicit. By using pointers, multi-dimensional array is declared as

data type (array name) [exp1][exp2]....[expn]

Or

data type * array name [exp1][exp2]....[expn]

eg:-

```
int (*p)[10];
```

⇒ p is a pointer to the int type two dimensional array.

29/8/18

eg:- (1)- int (*p) [10];

⇒ p is a pointer of the integer type and it is 2 dimensional.

(2)- int (*p) (char *a)

→ p is a pointer to a function which is taking pointer to character array "a" and returning integer value.

(3)- int (*p (char *a)) [10];

→ p is a pointer to a function taking character "a" as an argument and returning an integer array of 10 element.

(4)- int p (char (*a) [10]);

→ p is a function taking two-dimensional array "a" characters and returning an integer.

(5)- int p (char *a[]);

→ p is a function which is taking character array of pointers "a" as an argument and returning an integer.

(6)- int *(*p) (char (*a) [ ]);

→ p is a pointer to a function which is taking a two-dimensional character array "a" as an argument and returning pointer to integer.

(7)- int *(*p) (char *a[ ]);

→ p is a pointer to a function which is taking character array of pointer a as an argument and returning pointer to integer.

(8)- int (*p[10]) void ;

→ p is an array of 10 pointers to functions which is taking no arguments and returning an integer.

(9)- int * (* p[10]) (char a) ;

→ p is an array of 10 pointers to the function which is taking characters "a" as an argument and returning pointer to integer.

(10)- int * (* p[10]) (char *) ;

→ p is an array of 10 pointers to the function which is taking pointer of character arrays as an argument and returning pointer to integer value.

⇒ Objects and classes in C++ and Java / in OOP :

1. Access Specifier : public, private or protected.

- public → functions / member functions
- private → attributes, data variables
- protected → inheritance property.

eg:-

```
C++
class small
{
  private  :  int x;
  public   :  void setdata (int d)
              {
                x = d;
              }
              void display ();
              {
                cout << "\n data is " << x << endl;
              }
};
```

⇒ Structures in OOP :

Structures represents a no. of elements of different data type. It's general form is :

```
struct tag
{
  member 1 ;
  member 2 ;
    :
  member n ;
};
```

→ tag is optional which specifies the name of the structures.

(i)- No storage class is assigned to a structure.

(ii)- Members of a structure can not be initialized.

(iii)- A composition of a structured variable is defined as :

```
struct tag var1, var2, ... , varn ;
```

eg:-

```
struct part
{
  int model_no ;
  int part_no ;
  float cost ;
};
```

★ ⇒ Note : A structure can be a part of another structure.

eg:-

```
struct date
{ int month ;
  int day ;
  int year ;
};
```

```
struct account
{
    char acc^n_type ;
    int  acc^n_no. ;
    char name_holder [80] ;
    float balance ;
    struct date ⟶ last payment
};
```

⇒ Structures and pointers :

If we have a pointer variable ptvar which points to a structure of data type variable then

```
type * ptvar ;
ptvar = & variable ;
```

→ operator is used to access the member function of structure.

eg:-

```
type def. struc {
                    int acc^n_no;
                    char acc^n_type ;
                    char name [80] ;
                    float balance ;
                  };

account * pc;
pc = & acc^n_no ;
```

Ex:

```
type def. date {
                  int month ;
                  int day ;
                  int year ;
               };

type def account
{
    int * acc-no. ;
    char * acc_type ;
    char * name ;
```

float * balance ;

date

payment * pc ;

* pc = & account ;

* If we want to access acc^{n}_no by the structure account,

account . acc^{n}_no ;

(.) dot operator is used to assign / access members of a structure.

or if we want to access month then

date . month ;

(.) it has the highest precedence over other operators.

⇒ Passing structures into functions:

by values of member ;

by reference ;

by pointers ;

eg:-

```
typedef struct {
        int feet ;
        float inches ;
    } distance ;

distance add_length (distance , distance)
```

→ if we are passing function by values,

```
distance d1 , d2 ;
distance . add_length (d1 , d2)
{
    distance d3 = d1 ⊕ d2 ;
    return (d3) ;
}
```

this "⊕" has to be defined as we are using + in user defined data type

3/3/18

⇒ **Constructor:** A constructor in a program has following properties:

(i)- They do not return any values.
(ii)- They can create constant objects.
(iii)- They have the same name as class.

⇒ **Destructors:** A destructor in a program possesses following properties,

(i)- They do not return any value and do not take any arguments.
(ii)- They have the same name as the class names preceeded by "~" operator.
(iii)- They cannot be overloaded.
(iv)- They occur by default.
(v)- objects created in order are deleted in reverse order.

eg:-

```
class counter
{
    private:
        unsigned int count;
    public:
        counter()
        {
            count=0;
        }
    ~ counter() = { };
        void int count();
};
```

small * obj ptr :

It is a pointer to the object of type small.

small & S_1 ref :

implies that S_1 ref is the aliases to the object of type small.

S_1 ref = &S_1.

S_1 ref is used as aliases for S_1.

★

:: → Binary scope resolution operator.

eg)-

```
void counter :: int count()
{
    count++;
}
```

Q. Create a time class to display time in universal, and standard format, Standard time format is followed by AM or PM.

Soln :

```
#include <iostream>
#include <string>
class time
{ private:
        int hour;
        int minute;
        int seconds;
  public:
        Time();
        void settime(int, int, int) → default parameter passing
```

```
    void printUniversal();
    void printStandard();
};

Time::Time()
{
    hour = 0;
    minute = 0;
    seconds = 0;
}

Time::Time(int h, int m, int s)
{
    hour = h;
    minute = m;
    second = s;
};

void time::setTime(int h, int m, int s)
{
    hour = (h>=0 && h<24) ? h : 0;
    minute = (m>=0 && m<60) ? m : 0;
    second = (s>=0 && s<60) ? s : 0;
};
```

gives value

```
void time::printUniversal()
{
    cout<< setfill("0") << setw(2) << hour << ":" <<
    setw(2) << see minute << ":" << setw(2) <<
    second << endl;
};
```

```
void time :: printstandard ()
{
    cout << ((hour == 0 || hour == 12) ? 12 : hour %
    << ":" << setfill("*") << setw(2) << minute
    << ":" << setw(2) << second << (hour < 12) ? "AM"
    << endl;
};
int main ()
{
    Time t;
    t.printStandard();
    t.printUniversal();
    t.setTime(14,36,57);
    t.printUniversal();
    t.printStandard();
}
```

Output :

1. 00:00:00 AM
2. 00:00:00
3. 14:36:57
4. 02:36:57 PM

5/9/10

EX:

Constant Time = (6, 45, 0)

this cannot be modified by the program

Constant function { int gethour(); int getminute(); int getsecond(); } → Constant

→ Constant objects and constant functions :

↓ they can't be modified in the class / program

↓ functions which can not be modified.

Ex :- Back page

```
int time : sethour(h)
{
    hour = h;
}
```

✗ not allowed.

⇒ Utility function / utilities : Sometimes few functions are defined in the private part then such function are called as utility fn. when they are used to access the particular part of a program. In another words, utility fn's are the member functions of the particular part which assists the member function of the public part of a class.

Eg : Write a C++ program to input sales figure of 12 months and print out the annual sales report

Soln :

```
#include<iostream>
class salespersons
{
    private : double totalAnnualSale();
              double Sales[12];
    public : Salesperson();
```

```
    void getsalesfromuser();
    void getSales(int, double);
    void printAnnualSale();
};

SalesPersons : salesperson()
{
    for(int i=0; i<12; i++)
        sales[i] = 00;
}

void getsalesfromuser :: getsalesfromuser()
{
    double Salesfigure;
    for(int i=1; i<=12; i++)
    cout << "Enter the sales account for month"
         << i << "." << endl;
    cin >> sales[i];
      OR
    cin >> Salefigure;
        setvalues(i, salesfigure);

void set sales :: set sales (int month, double
                             amount);
{
    if (month > 0 && month<=12 && amount>=0)
        Sales[month-1] = amount;
    else
        cout << "Invalid month or salesfigure";
}
```

```
void SalesPerson :: printAnnualSale()
{
    cout << setprecision(2) << fixed << "Total Annual
    Sales are " << TotalAnnualSale() << endl;
}
```

[setprecision → it is used to set the no. of digits we want after decimal.
fixed → shows "." should also be present.]

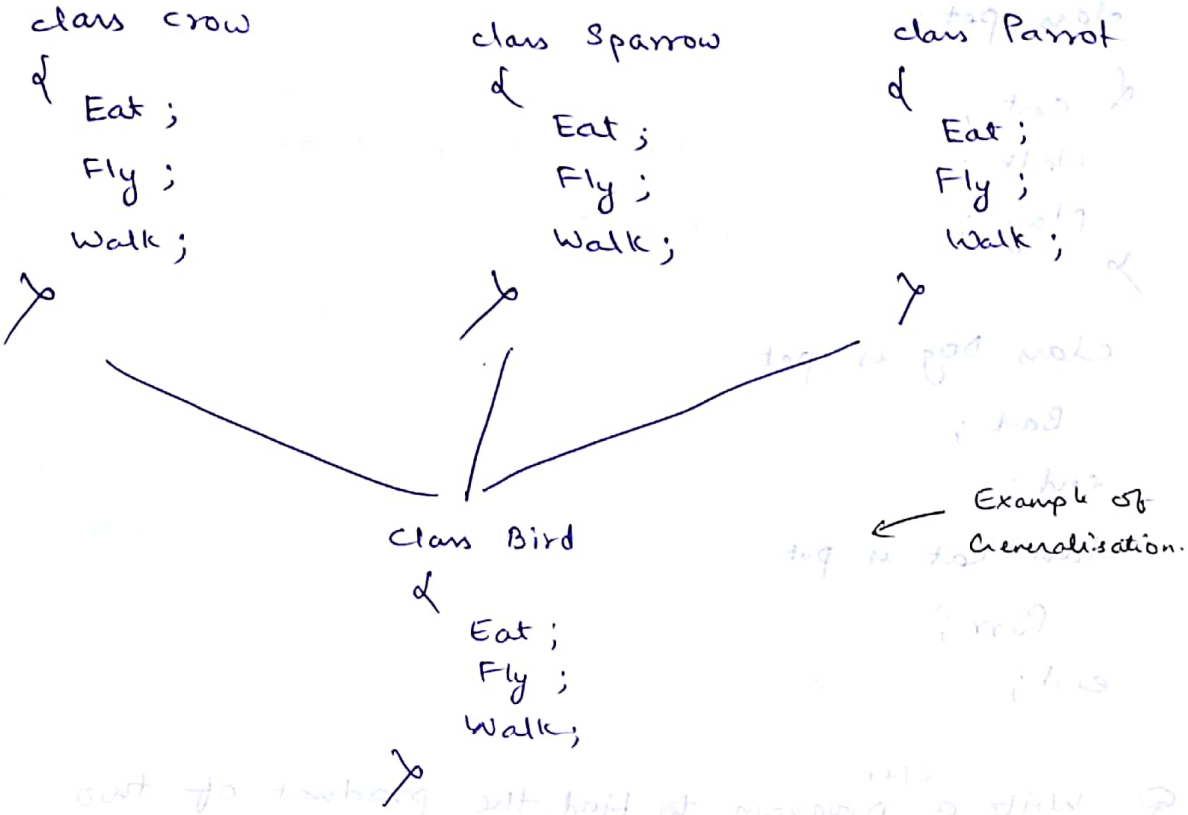
```
double SalesPerson :: TotalAnnualSale()
{
    double total = 0.0;
    for (int i=0; i<12; i++)
        total += Sales[i];
    return total;
}
```

private utility function

⇒ Generalizations :
It is the process of extracting shared characteristics from two or more classes and combining them into a generalized super class.

⇒ Aggregation : It is defined as a group of assembly and relationship (or function) between classes. (◇)

⇒ Inheritance : Many subclass can be derived from a superclass with the property that the derived class inherits the properties of the super class as well as they can define their own property or overwrite the existing properties of superclass. (△)



→ Ex. of Inheritance

```
class Dog
{
    walk ;
    eat ;
    play ;
    bark ;
}
class GermanShepherd
{
    walk ;
    eat ;
    play ;
    bark ;
}
```

Ex. of Aggregation

```
class Dog
{
    walk ;
    eat ;
    play ;
    bark ;
}
class Cat hasDog
{
    Eat = Dog.eat ;
    play = Dog.play ;
    walk = Dog.walk ;
    Pun ;
}
end ;
```

```
class pet
{
  eat;
  walk;
  play;
}

class Dog is pet
  Bark;
end;
class Cat is pet
  Purr;
end;
```

Q. Write a C/C++ program to find the product of two matrices i.e. $[A]_{m \times n} \times [B]_{n \times p} = [C]_{m \times p}$

→

```
#include <stdio.h>
main()
{
  int i, m, n, p, q, a[10][10], b[10][10], c[10][10], j;
  printf("Enter the order of A and B respectively:");
  scanf("%d %d %d %d", &m, &n, &q, &p);
  printf("The matrix A is \n");
    for(i=0; i<m; i++)
      for(j=0; j<n; j++)
          scanf("%d", &a[i][j]);
  Similarly B.   (i=0, i<q)
                 & (j=0, j<p)

  if (n==q)
  {
    for(i=0; i<m; i++)
      for(j=0; j<p; j++)
```

```
{
    s=0;
    for(k=0; k<n; k++)
        s+= a[i][k]*b[k][j];
    c[i][j] = s;
}
}
else
{
    print(" Matrix Multipli. is not possible");
    printf(" Product (A & B) is \n");
    for(i=0; i<m; i++)
    {
        for(j=0; j<p; j++)
            printf("%d", &c[i][j]);
    }
}
```

```
Struc date ;
{
    int month ;
    int day ;
    int year ;
};
```

```
Structure Acc^n ;
{
    char acc_type;
    char acc_no.. ;
    char name[10];
    date ;
};
```

⇒ **Self-referential structure :**

If a member of the structure points to the same structure, then they are called self-referential structures.

Ex :-

```
Struct Employee
{
    member 1 ;
    member 2 ;
    :
    :
    Struct employee * name;
};
```

⇒ Constructors & Destructors :

(i)- Do not return any values.

(ii)- They can create constant object.

(iii)- They may have same name as class.

Dest

(iv)- Do not return any value

(ii)- They may have the same name as the class proceeded by ~.

(iii)- They cannot be overloaded.

(iv)- Objects created in order are deleted in reverse order.

Ex :

Examples of Constructors & Destructors :

Soln :

```
# include <iostream.h>
# include <string.h>

class createanddestroy
{
  public :
     createanddestroy (int, string);
     ~ createanddestroy ( );
  private :
     int objectID;
     String massage;
};
```

```
Createanddestroy (int ID, string messagestring)
{
    Object ID = ID;
    message = messagestring;
    cout << "Object" << Object ID << " Constructor-runs"
         << message << endl;
};

createanddestroy.~Create and destroy()
{
    cout << (Object ID = 1 || Object ID == 7 ? "\n" : " ");
    cout << "Object" << Object ID << "destructor-runs" << message
         << endl;
}

int main()
{
    void create(void);
    create anddestroy (1, "global before main");
    cout << "main function execution begins" << endl;
    createanddestroy (2, "local automatic in main");
    static createanddestroy (3, "local static in main");
    create();
    cout << "main function resumes" << endl;
    Create and destroy fourth (4, "local automatic in main");
    cout << "main function ends" << endl;
    return();
}
void create(void)
{
    cout << "Create function execution begins" << endl;
```

```
    Create and display fifth (5, "local automatic in main");
    Static create and display sixth (6, local static in main);
    cout << " create function ends" << endl;
};
```

output is :

1. Object 1 Constructor runs (global before main)
2. main function begins
3. Object 2 Constructor - runs (local automatic in main)
4. Object 3 " " (local automatic in main)

(
|
|

⇒ Polymorphism : (different forms)

It means a function when applied to different objects, it gives different attributes of that objects.

eg:-

(1). draw() → which draws and calculates areas of geometric figures is an example of polymorphism.

⇒ Polymorphism has two features :

(1)- Operator overloading.

(2)- Function Overloading.

(1)- Operator Overloading : Operators are overloaded to define new behaviour of objects depending on the types of operands (<<, >>).

eg:- Using Operator Overloading use "+" to add two points in polar-coordinates system.

$[P_1(r_1, \theta_1) \& P_2(r_2, \theta_2)]$

(ii)- Function Overloading :

Same function name is used for different purposes depending on the arguments.

eg:- draw()

{

for (i=0 ; i<= number of shapes ; i++)

area of shape[i] = area of shape[i].draw();

→ Data conversions :

It can be of different types :

(i) - Basic Type

(ii) - User-defined type

(iii) - Both basic and user-defined type.

(Conversions)