BCS-054 Object Oriented System Design with C++

BTECH CSE – Vth Semester

Software Development in C++

Software Development Process which is same process as C and C++ language

First you create .EXE file but this becomes problem to create the file because C++ source code understandable for Operating System

So, there are similar Operating System in Computers For example : Windows DOS, Solaris, MAC so this exe file which is developed for a particular operating system

Solaris is a Unix-like operating system that was originally developed by Sun Microsystems

While if we learn C++ Language the problem occurs that we have to develop similar exe file for others operating system

So we don't create EXE file directly so generally we learn these thing in C++ means whatever we want to develop these thing we do in C++ language

In .cpp file there are some code which starts from #include<iostream.h> and this statement resolves after reading by Preprocessor

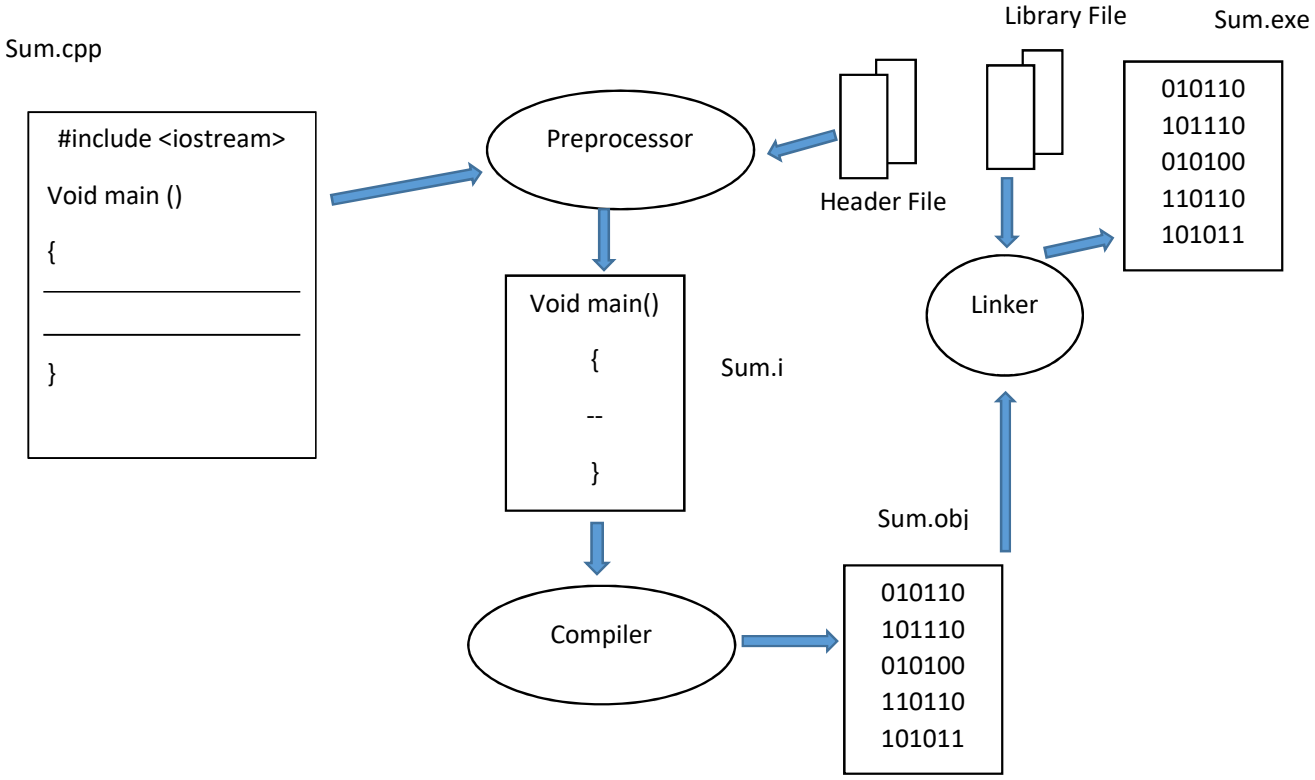
And #include<iostream.h>

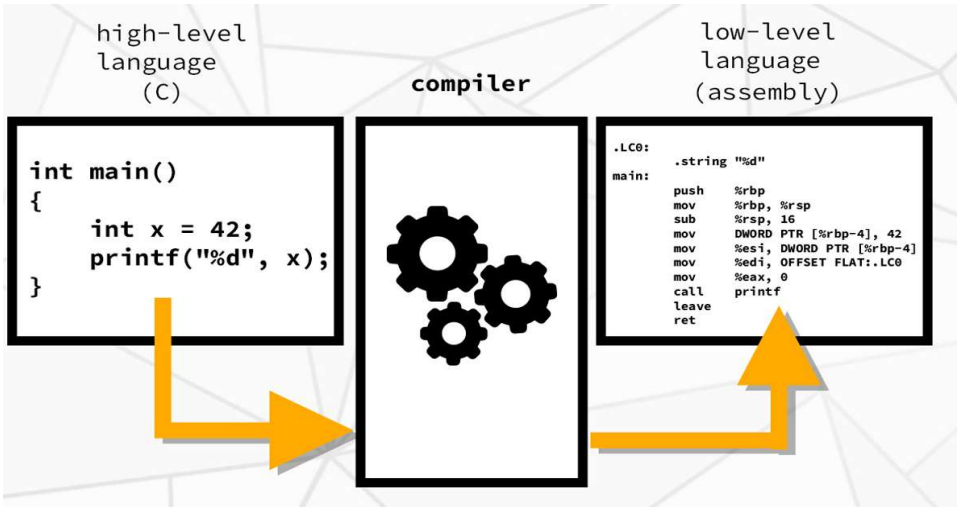
1- First we write CPP file
2- We require Preprocessor software
3- Preprocessor file Link from our File to Header File
4- Our file checks #include<iostream.h>
5- In Our File Preprocessor works to resolve the statement
6- It creates another file sum.i after resolve sum.i #include<iostream> without .h means no any statement which is #include .h and our file includes Header file's content
7- Sum.i creates in RAM but not save any where
8- Now compiler software read this Sum.i file
9- Compilers are computer programs that translate high-level programming code into lower-level code like Machine Language
10- Compiler convert the whole program into machine code (called Object code) and it Generates Object File this is understandable for operating system

11- While operating System which does not understand the code of Header's content which is taken from Library Header file

12- Now Linker software require which object file contents is linked to Library file's content and creates new file i.e, .exe file

13- Exe File is a software





High Level -> Compiler - > Machine Level Language-> (Object code)-> Binary code

And as we are mentioned Header file so that content of Header file will replace in .cpp file which line starts from #include

Now it creates New separate file and Preprocessor has removed the statement of #include line in this file our code is already given while it is also replaced header file content. and it has resolved it and this file is Sum.i which does not save it while it creates in RAM

This Sum.i file will read by Compiler software which works for this reading and translate into object code which will create intermediate file and it will be understandable language by operating System which we know **Object file**

And in starting we have created .cpp file that is source file that is not any software and no any operating system can run it

That's why with the help of preprocessor and compiler an object file is generated and compiler has translated and converted this code for Operating System understandable language

But this Sum.obj file which is some code is not understandable for operating system because some code is written already into the Library File.

Library Files's code is already into the Sum.obj

Now we need an another software which is called as Linker which link the content of Object File Sum.obj with Library File's content and it creates new file i.e, Sum.exe

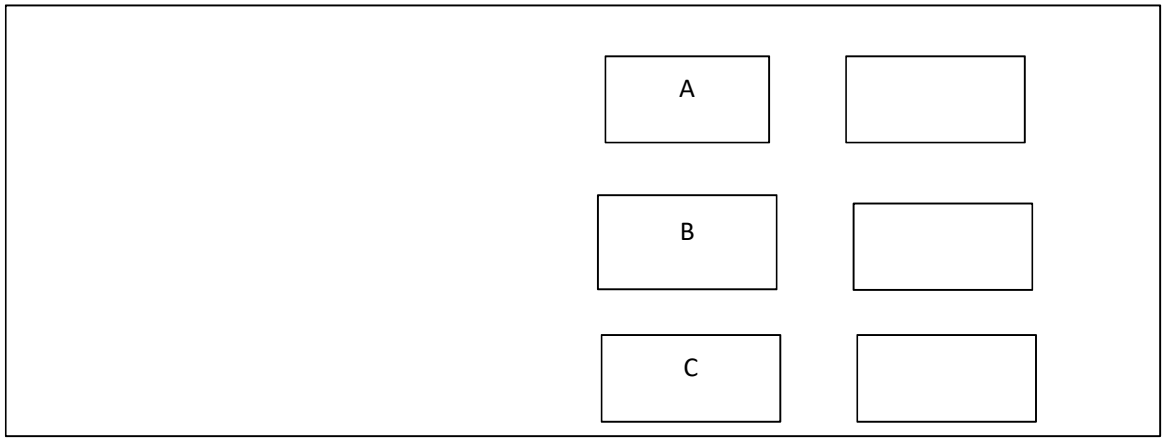
Next Class Lecture

Constant ->Information is constant Data = Information = Constant

Primary Constant	Secondary Constant
Int : 23, -341, 0,	Array
Real : 120.45	String : "Gaurav"
Character : "a", "+","2"	Pointer
	Structure
	Class

Variable
Memory Management
In Program two things important

1> Data
2> Instruction



Variable can not starts from digit value

Keyword

break

Auto

Case

Const

Continue

If

For

Through

This

Data Types

int a,b=5; 2 byte integer constant

char ch =’a’; 1 Byte character constant

float k=3.45; 4 Byte real constant

double d1 8 Byte real constant

C and C++

Input/Output in C++

In C language we use printf() function and in C++ we use cout

Cout is a predefined Object

In c Language uses format specifier %d

The operator << (insertion operator) use in this

Cin is also predefined Object and the operator >> Extraction or get from operator use in this

printf("The sum of %d",a,"and %d",b,"=%d",c);

cout<<"sum of"<<a <<"and "<<b <<"is "<<c;

we can aslo use scanf function use in C++

```
#include <iostream>
#include <stdio.h>
using namespace std;
int main() {
int a,b,sum;
cout << "Enter any two integers: ";
cin >> a;
  scanf("%d",&b);
  sum=a+b;
// sum of two numbers in stored in variable sumOfTwoNumbers
//sum = first_number + second_number;
// prints sum
cout <<"Total value of "<<a <<" and "<<b <<" is "<<sum;
return 0;
}
```

About iostream.h

We need to include Header File iostream.h, it contains a declaration for identifier cout and the operator << and also for the identifier cin and operator >>

The header file contains a declaration of identifiers

Identifier can be function name, variable, object, macros, etc.

End

endl is a manipulator and declared iostream.h

Identifier

In C++ an identifier is a unique name used to identify variable, function, classes, structure and other entities in a program

int money;

double accbalance;

Here money and accbalanace are identifier

It can only consist of letters (**uppercase and lowercase**), digits, and underscores.

It must begin with a letter or an underscore. It cannot be a digit.

They are case-sensitive, which means uppercase and lowercase letters are considered different characters. E.g., "myVar" and "myvar" are two different identifiers.

They cannot be the same as any reserved words in the C++ language. Examples of reserved words in C++ include "if", "while", and "for".

Identifiers cannot contain any spaces or special characters such as @, #, $, %, ^, &, *, etc.

Constant

Constants refer to fixed values that the program may not alter and they are called **literals**.

Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

Type Casting in C++

This section will discuss the type casting of the variables in the C++ programming language. Type casting refers to the conversion of one data type to another in a program. Typecasting can be done in two ways: automatically by the compiler and manually by the programmer or user. Type Casting is also known as Type Conversion.

Type of Type casting :

Implicit Type casting

it is done by the compiler and there is no loss of information

x	Y	x+y

8 bit	16 bit	Now conversion will be 16 bit
16-bit		

For instance, the ASCII code for the uppercase letter “A” is 65, while the code for the lowercase letter “a” is 97

```
#include <iostream>
using namespace std;
int main ()
{
   short x = 32770;
   int y;
   y = x;
   cout << "sizeof short datatype is: " << sizeof(short)<<endl;
   cout << " Implicit Type Casting " << endl;
   cout << " The value of x: " << x << endl;
   cout << " The value of y: " << y << endl;

   int num = 20;
   char ch = 'a';
   int res = 20 + 'a';
   cout << " Type casting char to int data type ('a' to 20): " << res << endl;
   float val = num + 'A';
   cout << " Type casting from int data to float type: " << val << endl;
   return 0;
}
```

output:

sizeof short datatype is 2

The value of x: 200

The value of y: 200

Type casting char to int data type ('a' to 20): 117

Type casting from int data to float type: 85

Explicit Type casting : It is done by the programmer and there will be loss of information

it is done

```
#include <iostream>
using namespace std;
int main ()
{
   float x = 5.445;
   int y;
   y = (int)x;
```

```
        cout<<"Y Value "<<y;
    return 0;
}
```

Ex: Adding two string :

```
#include <iostream>
#include <string.h>
using namespace std;
int main ()
{
  string m ="10", n="42";
  cout<<m+n;
  return 0;
}
```

Example :

```
#include <iostream>
//#include <string.h>
using namespace std;
int main()
{
    char ch='a';
    int m=45+ch;
    char a[1]={'b'};
    int z=40 + a[0];
    cout<<m<<'\n'; //142
    cout<<z<<'\n';  //138
  return 0;
}
```

Ex: Adding two Number in string type and convert into string :

```
#include <iostream>
#include <string.h>
using namespace std;
int main ()
{
  string a ="10", b="12";
  int i=stoi(a), j=stoi(b);
  int sum=i+j;
string ans1=to_string(i);
string ans2=to_string(j);
string z=ans1+ans2;
```

```
cout<<sum;
cout<<z;
  return 0;
}
```

Output :
22
1012

Example

```
int main()
{
  string a ="26", b="34";
  int i=stoi(a), j=stoi(b);
  string sum=a+b;          //2634
  cout<<sum<<'\n';         //2634
  string k=to_string(i);   //26
  string g=to_string(j);   //34
  string f=k+g;
  cout<<f<<'\n';           2634

  return 0;
```

Example :

Although both expressions can be used to create a variable to store one character, there are following differences.

1) “char a” represents a character variable and “char a[1]” represents a char array of size 1.

2) If we print value of char a, we get ASCII value of the character (if %d is used). And if we print value of char a[1], we get address of the only element in array.

```
#include <stdio.h>


int main ()
{
  char a1 = 'A';
  char a2[1] = {'A'};
  printf("%d  %d", a1, a2);
  return 0;
}
```

Example

```
#include <iostream>
#include <string>
using namespace std;
int main() {
char c;
cout << "Enter a character: ";
cin >> c;
int asciiValue = c;
cout << "Character Value " << c << endl;
cout << "The ASCII value " << c << " is " << asciiValue << endl;
return 0;
}
```

Output :

Enter a character: a

Character Value a

The ASCII value a is 97

Example :

Typedef :

typedef keyword in C++ is used for aliasing existing data types, user-defined data types Typedefs allow you to give descriptive names to standard data types,

Syntax:

```
typedef <current_name> <new_name>
```

```
#include <iostream>
using namespace std;

int main()
{
   typedef int grv;

   grv a = 15; //a is int type internally
   grv b = 45; //b is int type internally
   grv c=a+b;
   cout << "Both Number "<< "\n";
   cout << "Total " << c << "\n";
   return 0;
}
```

Example

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  struct {
    int myNum;
    string myString;
  } ashoka;

  ashoka.myNum = 1;
  ashoka.myString = "Welcome Ashoka";

  cout << ashoka.myNum << "\n";
  cout << ashoka.myString << "\n";
  return 0;
}
typedef struct employee {
int eid;
```

```
char favchar;
int salary;
} ep;

int main()
{
ep hurry;
struct employee shubham;
struct employee sahil;
harry.eid=4;
harry.favchar='s';
harry.salary=15000;
cout <<"The Value"<< harry.eid <<endl;
cout <<"The favchar"<< harry.favchar <<endl;
cout<<"Salary is "<<harry.salary<<endl;

return 0;
}
```

Example :

```
// C++ Program to showcase the use of typedef
//  with data pointer

#include <iostream>
using namespace std;

int main()
{
   int a = 10;
   int b = 40;
   // iPtr can now be used to create new pointers of type
   // int
   typedef int* iPtr;

   iPtr ptr_to_a = &a;
   iPtr ptr_to_b = &b;
```

```
    cout << "a is: " << *ptr_to_a << "\n";
    cout << "b is: " << *ptr_to_b << "\n";

    return 0;
}
```

Output:

```
a is: 1
b is: 40
```

Q : What is the structure of the C++ program?

Ans : In C++, a program divides into three sections:

1. Standard Libraries Section
2. Main Function Section
3. Function Body Section.

Structure:

```
struct {
  int myNum;
  string myString;
} myStructure;

// Assign values to members of myStructure
myStructure.myNum = 1;
myStructure.myString = "Hello World!";

// Print members of myStructure
cout << myStructure.myNum << "\n";
cout << myStructure.myString << "\n";
```

Program Structure :

Square program in C++

```
#include<iostream>
using namespace std;
  int main() {
  int x;
  cout<<"Enter your any Number";
  cin>>x;
  int s=x*x;
```

```
  cout<<"Square of "<<x <<" is "<<s;
  return 0;
}
```

Control Structure :

Many times in our code, we will need that a particular piece of code should only execute if a specific condition gets fulfilled. In C++, along with Conditional Control Structures, we also have the Iteration or Loop Control Structure.

Types of Control Structure in C++

There are three types of Control Structures in C++, And all the program processes we write can only be implemented with these three control structures.

Sequence Structure

This is the most simple and basic form of a control structure. It is simply the plain logic we write; it only has simple linear instructions, no decision making, and no loop. The programs we all wrote at the start of our programming journey were this control structure only. It executes linearly line by line in a straight line manner.

Selection Structure

Sometimes in our program, we will need to write certain condition checks that this part of the code should only execute if a particular condition meets or another part of code should run.

```
#include"iostream"
using namespace std;
int main()
{
	int f=0,s=0,t=0;
	cout<<"Enter First Marks  ";
	cin>>f;
	cout<<"Enter Second Marks  ";
	cin>>s;
	cout<<"Enter third Marks  ";
```

```
	cin>>t;
	if(f > s && f > t)
		cout<<"First Values "<<f<<" bigger"<<'\n';
	if(s > f && s > t)
		cout<<"Second Values "<<s<<" bigger"<<'\n';
	if(t > f && t > s)
		cout<<"Third Values "<<t<<" bigger"<<'\n';

return 0;
}
```

Example

```
#include"iostream"
using namespace std;
int main()
{
	int num=0;
	cout<<"Enter Number ";
	cin>>num;
	if(num > 0)
		cout<<"Number is Positive "<<'\n';
	else if(num < 0)
		cout<<"Number is Negative "<<'\n';
	else
		cout<<"Number is "<<num<<'\n';

return 0;
}
```

Loop Structure

Suppose we are asked to write a program that prints "Hello World" once. We will simply write it as:

```
 cout << "Hello World";
```

Now, we are asked to write a program that will print "Hello World" 10 times; we can write it as

```
int main() {
```

```
    cout << "Hello World" << endl; // 1 time
    cout << "Hello World" << endl; // 2 time
    cout << "Hello World" << endl; // ..
    cout << "Hello World" << endl; // ..
    cout << "Hello World" << endl; // 5 time
}
```

Now, what if we have to print it 100 or, let's say, 1000 times? Are we going to copy-paste the same line 1000 times? No, that will not be feasible. There comes the use of loop control structures.

Whenever in our program we see that a certain piece of code is being repeated repeatedly, then we can enclose that in a loop structure and provide the condition that this code should execute these specific number of times.

Example

```
#include<iostream>
using namespace std;

int main()
{
	int num=0;
	cout<<"Enter Number ";
	cin>>num;
	for(int i=0;i<num;i++)
		cout<<"Ashoka College"<<'\n';

return 0;
}
```

We have three types of Loops in C++:

While Loop
Do While Loop
For Loop

While Loop

```
#include"iostream"
using namespace std;
int main()
{
int num,fact=1,i=1;
```

```
   cout<<"\n Enter The Number:";
   cin>>num;

   while(i<=num)
   {
      fact=fact*i;
      i++;
   }
 cout<<"\n The Factorial of "<<num<<" is "<<fact;


return 0;
}
```

Control statements in C/C++ to Implement Control Structures
Conditional Statements
In general, we have two types of conditional statements in C++:

if statements
if else-if ladder

Iteration Statements
As discussed above, sometimes, in our program, when we want to repeat a certain set of instructions, we can wrap them inside a loop. In C++, we have three types of iteration or loop statements:

While loop
Do while loop
For loop

Jump Statements
As the name suggests, jump statements are used to jump out of a block of code, be it a conditional statement, an iteration statement, or something else. In C++, we generally have 4 types of jump statements:

break
continue
goto
exit()

Switch Statements

The switch statement in C++ is a flow control statement, The C++ Switch case statement evaluates a given expression and based on the evaluated value(matching a certain condition)

Syntax:

```
switch(expression)
{
   case value1:
      // statements to run
      break;
   case value2:
      // statements to run
      break;
   ..
   ..
   ..
   default:
      // statements to run
}
```

Friend Function

Friend Function is not member function of a class to which it is a friend

Friend function is declared in the class with friend keyword

It must be defined outside the class to which it is friend

Friend Function can become friend to more than one class

Friend Access	Normal Function
It can access any member of class while if it is private, public, protected	While function can not access the private data member directly

Friend Function is declared in the class with the friend keyword
Friend Function can access any member of a class to which it is friend
Friend function c an not access member of the class directly
it has no caller object
it should not be defined with membership label

Example :

```
class Complex
{
	int a,b;
	public:
		void setData(int x,int y)
		{
			a=x;
			b=y;
		}
		void showData()
		{
			cout<<"Sum "<<a<<"b "<<b;
		}

		friend void fun(Complex); //now declrae the friend function
};
void fun(Complex c)
{
	cout<<"Sum is "<<c.a+c.b<<'\n';
}
int main()
{
	Complex c1,c2;
	c1.setData(41,23);
	fun(c1);
	return 0;
}
```

Output :
Sum is 64

Example :

```
#include"iostream"
using namespace std;
class B;
class A
{
	int a;
	public:
		void setData(int x)
		{
			a=x;
		}
	friend void fun(A,B);
};


class B
{
	private:
		int b;
	public:
		void setData(int y)
		{
		b=y;
		}
	friend void fun(A,B);
};
void fun(A o1,B o2)
{
	cout<<"sum is "<<o1.a+o2.b<<"\n";
}
int main()
{
	A obj1;
	B obj2;
	obj1.setData(41);
	obj2.setData(31);
	fun(obj1,obj2);
```

```
	return 0;

}
```

Output :
sum is 72

Operator overloading
Example

```
// C++ Program to showcase the use of typedef
//  with data pointer

#include <iostream>
using namespace std;
class Complex
{
	int a,b;
	public:
	void setData(int x,int y)
	{
		a=x;
		b=y;
	}
	void showData(){
		cout<<"\n"<<a<<"\n"<<b;
	}

	Complex operator+(Complex c)// declrae friend function
	{
		Complex temp;
		temp.a=a+c.a; //a belongs to caller object in main
function c1
		temp.b=b+c.b; //b belongs to caller object in main
function c1
		return (temp);
	}
```

```
};

int main()
{
	Complex c1,c2,c3;
	c1.setData(4,6);
	c2.setData(12,18);
	c3=c1+c2;  //c3=c1.operator+(c2)
	c3.showData();
	return 0;
}
```

Output :
16
24

Overloading of operator as a Friend function

```
#include <iostream>
using namespace std;
class Complex
{
	int a,b;
	public:
	void setData(int x,int y)
	{
		a=x;
		b=y;
	}
	void showData(){
		cout<<"\n"<<a<<"\n"<<b;
	}
	friend Complex operator+(Complex,Complex);  // declrae friend
function

};
	Complex operator+(Complex X,Complex Y)// difinition friend
function
	{
```

```
            Complex temp;
            temp.a=X.a+Y.a; //a belongs to caller object in main
function c1
            temp.b=X.b+Y.b; //b belongs to caller object in main
function c1
            return (temp);
      }


int main()
{
      Complex c1,c2,c3;
      c1.setData(4,6);
      c2.setData(12,18);
      c3=c1+c2;  //c3=operator+(c1,c2);
      c3.showData();
      return 0;
}
```

Output:
16
24

Data Type :

Example: 1.9876, 5.43219, -876.543, 21.987654, 0.15197e-7, etc
A double data type can store floating-point values up to 15 digits

Size of data type

```
#include <iostream>
using namespace std;
int main()
{
   int integerType;
   char charType;
   float floatType;
   double doubleType;

   cout << "Size of int is: " << sizeof(integerType) << "\n";

   cout << "Size of char is: " << sizeof(charType) << "\n";
```

```
    cout << "Size of float is: " << sizeof(floatType) << "\n";

    cout << "Size of double is: " << sizeof(doubleType) << "\n";

    return 0;
}
```

Output :

```
Size of int is: 4
Size of char is: 1
Size of float is: 4
Size of double is: 8
//Double variables typically require 8 bytes of memory space.
```

some time int was 2 Byte when the machine had processor and instruction register was 16 bit. meas at a time it can store zero in 16 times now it became 16 bit system
and in 90 decade it became 32 bit and stored 32 bit instruction register

```
#include <iostream>
using namespace std;
int main()
{
    char x[3]={'A','B'};
    int m =int(x[0]);
    cout << "Size of int is: " << sizeof(m) << "\n";
    return 0;
}
```

Output :
Size of int is 4
Example :

```
int main()
{
    int x[3]={3,2,6};
```

```
    cout << "Size of int is: " << sizeof(x) << "\n";


    return 0;
}
```

Output :
Size of int is : 12

Example:

```
int main() {
        int m=0;
  int x[5] = {10, 20, 30, 40, 50};
  int k=sizeof(x)/sizeof(x[0]);

  cout<<"K Value "<<k;
  return 0;
}
```

Output
K value 5
Example :

```
#include <iostream>
using namespace std;

int main() {
        int m=0;
  int x[5] = {10, 20, 30, 40, 50};
  for (int i=0;i<5;i++)
  {
        m=m+sizeof(x[i]);
    cout <<m << "\n";
  }
  return 0;
}
```

Output:
4 8 12 16 20

Example :

```
int main() {
       int m=0;
  int x[5] = {10, 20, 30, 40, 50};
  int k=sizeof(x)/x[0];
  for (int i=0;i<sizeof(x)/sizeof(x[0]);i++)
  {
       m=m+sizeof(x[i]);
    cout <<m << " ";
  }
  return 0;
}
```

Output :
4 8 12 16 20

Example :

```
#include <iostream>
using namespace std;
int main()
{
   char x[3]={'a','b','c'};
   cout << "Size of char is: " << sizeof(x) << "\n";


   return 0;
}
```

Output : 3

Example float size

```
#include <iostream>
using namespace std;
int main()
{
   float x[2]={15.4,1.6};
```

```
    cout << "Size of float is: " << sizeof(x) << "\n";
   return 0;
}
```

Output :
Size of float is: 8

Inline Function

The inline function is not suitable for big program
Defined inside the class
works as macros
Does not allow control statement
function declared in class and definition is the out of the class
syntax : inline return_type class name ::function name
it replce the function definition code in the line of function calling so it
when we're using inline function then this is like request for complier

Example:

```
#include <iostream>
using namespace std;
class Add
{
      int a,b;
      public:
      int addition(int x, int y);
};
      inline int Add::addition(int x, int y)
      {
      return(x+y);
      }
int main()
{
      Add a1;
      int c=a1.addition(12,34);
      cout<<"Sum is "<<c<<endl;
      return 0;
}
```

Output:
Sum is 46

Outline Function

defined outside the class
work as normal member function

syntax : return_type class_name :: function name

Example

```
#include <iostream>
using namespace std;

class person
{
	private:
	char name[10];
	int age;
	public:
		void input();
		void output();

};
	void person::input() //outline function
		{
			cout<<"\nEnter Name ";
			cin>>name;
			cout<<"\nEnter age ";
			cin>>age;
		}
	void person::output() //outline function
		{
			cout<<"Name "<<" "<<name;
			cout<<"\nAge "<<" "<<age;
		}

int main()
{
	person p;
	p.input();
	p.output();
	return 0;
```

}

Output:
Enter Name ashoka
Enter Age 18
Name ashoka
Age 18

The compiler may not perform inlining in such case circumstances like
if a function contains loop(for while do while)
if a function contains static variable
if a function contains recursive

Outline function
class persone
{
char name

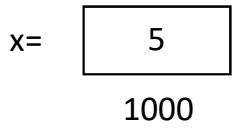
Reference Variable

When a variable is declared as a reference, it becomes an alternative name for an existing variable.

In C++ Reference Variable declared with data type

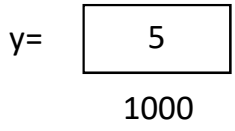
x=5;

int &y=x;



x=5;

int &y=x;



y++

cout>>y;

output : 6

it can be initialized with already declared variable only

Reference variable can not be updated.

Global Variable

```
#include <iostream>
using namespace std;
int k;
int main()
{
	cout <<"Enter two Number "<<endl;
	int n1,n2,n3;
	cin>>n1 >> n2;
	k=n1+n2;
	cout<<"The sum of "<<n1 <<"and "<<n2 <<" is "<<k<<endl;

	return 0;
}
```

Input String

```
#include <iostream>

using namespace std;

int main()
{
	cout <<"Enter any Name "<<endl;
	string n1;
	cin>>n1;
	cout<<"The name is "<<n1 <<endl;
	return 0;
}
```

Output :

Gaurav Singh

The name is Gaurav

Concatenate String

```
#include <iostream>
using namespace std;
int main()
{     string s="GAURAV";
      string h ="Kumar";
      cout <<"Enter any full Name "<<endl;
      //getline(cin,s);
      cout<<"The Name is "<<s +" "+ h;
      return 0;
}
```

Output

The Name is Gaurav Kumar

Using strcat function for Concatenation

When strcat uses then apply header file #include<cstring>

```
#include <iostream>
#include <cstring>
using namespace std;

int main()
{     char a[10],b[16];
      cout<<"Enter First Value "<<endl;
      cin>>a;
      cout<<"Enter Second Name "<<endl;
      cin>>b;
      strcat(a,b);
      cout<<a;
      return 0;
}
```

Input String and output with space

when working with strings, we often use the getline() function to read a line of text. It takes cin as the first parameter, and the string variable as second:

```
#include <iostream>
using namespace std;

int main()
{     string s;
      cout <<"Enter your any Name "<<endl;
      getline(cin,s);
      cout<<"The Name is "<<s;
      return 0;
}
```

Pre increment and Post Increment

```
int a=1;
//cout<<a++ //  assign only value in variable 1
cout<<++a // increment by 1 then assign it
```

What is Function

Function is a block of code and set of instruction performing a unit task.

Function has a name return type and arguments.

Functions are predefined and user defined

Function are generally Two types

Predefined Function and User Defined Function

Pre-Defined function like printf() scanf() getch() clrscr() which definition is given in Library file while its declaration given in Header file

What are the two ways to pass arguments to the function

Call by value: This method copies the value of an argument into the formal

parameter of the function.

Call by reference: This method copies the address of an argument into the formal

Function Declration, Declaration and Call

```
#include <iostream>

using namespace std;

int main()

{     void fun(); //Function Declare

      cout<<"You are in Main"<<endl;

      fun(); //Function Calling

      return 0;

}

void fun() //Function Definition
{
      cout<<"You are in Fun Function ";
}
```

Output :

You are in Main
You are in Fun Function

As a Function declared in the Main Function, this kind of function is called local declaration only the main function can call this Fun Function

If we want that fun function can called by any function in whole program than declaration of fun function should be globally means Then we can place void fun(); before the int main() function

Function apply generally Takes Something Return Something
Function Also Generally Takes Something Return Nothing
Takes Nothing Return Nothing
Takes Nothing Return Something

Example :1

```
#include <iostream>
using namespace std;

int sum(int,int);  //Function declare
int main()
{
	int a=12,b=34;
	int s=sum(a,b); // actual argument
	cout<<"Total Value "<<s;
	return 0;
}
int sum(int x, int y) //formal argument
{
	return(x+y);
}
```

Output:

Total Value 46

Example

```
#include <iostream>
using namespace std;
int main()
{
	cout<<"This is Main Program";
	void sum();  //Function declare
	sum();
	return 0;
}
void sum()
{	int a=41,b=34;
	int z=a+b;
	cout<<"\nTotal Value = "<<z;
}
```

Types of Formal Arguments

Which we define in Function, Formal arguments can be of three types

- Ordinary Variable of any Type

- Pointer Variable
- Reference Variable

Call by Value / Function call by passing Value

When formal arguments are ordinary variable, it is a function call by value

```
#include <iostream>
using namespace std;

int sum(int,int);  //Function declare
int main()
{
	int a=12,b=34;
	int s=sum(a,b);
	cout<<"Total Value "<<s;
	return 0;
}
int sum(int x, int y) // arguments are ordinary variable
{
	return(x+y);
}
```

Example 2

```
int main()
{
void sum();
	sum();
return 0;
}

void sum()
{	int a=41,b=34;
	int z=a+b;
	cout<<"\nTotal Value = "<<z;
}
```

Call by Address

When formal arguments are pointer variable it is function call by address

Here in the main program function is call and passing with the address of a and address of b means the address of a and address of b are passing in corresponding *p and *q

Which is pointer pointing variable a and variable q

Example

```
#include <iostream>

using namespace std;

void bigger(int *,int *);

int main()

{

	int a=51,b=34;

	bigger(&a,&b);

	return 0;

}

void bigger(int *x, int *y)

{

	if (*x > *y)

		cout<<"First Value is Bigger "<<*x;

	else

		cout<<"Second Value is Bigger "<<*y;

}
```

Output

First Value is Bigger 51

Call by Reference

When formal arguments are reference variable, it is function call by reference

Example

```
#include <iostream>
using namespace std;
void bigger(int &,int &);
int main()
{
	int a=51,b=35;
	bigger(a,b);
	return 0;
}
void bigger(int &x, int &y)
{
	if (x > y)
		cout<<"New First Value is Bigger "<<x;
	else
		cout<<"New Second Value is Bigger "<<y;
}
```

Output

First Value is Bigger 51

Inline Function

Inline is a request not a command

Complier checks how much memory consuming after writing inline during the function declaration

The benefit of inline function when this is small size of function and it reduces function grows in size

So the compiler may ignore the request in some situations.

Few of them Time 7:51 Lecture 5 in C++ part 2

A function containing loops switch goto

Function with recursive

Containing static variable

The above line compiler can produce a warning when rec. loop switch static variable used

---

Total Marks calculate by Function

```
#include <iostream>
using namespace std;
int sum(int [],int);
//int compute_grade([]);
int main()
{	int i,n,total;
	//char g[2];
	int arr[10];
	cout<<"Enter No. of Subject "<<endl;
	cin>>n;
	cout<<"Enter Marks "<<endl;
	for(i=0;i<n;i++)
		cin>>arr[i];
	total=sum(arr,n);
	cout<<"Total Marks = "<<total;
	return 0;
}
int sum(int m[],int s)
	{
		int i,sum=0;
		for(i=0;i<s;i++)
			sum=sum+m[i];
		return sum;
	}
```

Example

```
#include <iostream>
using namespace std;
//int sum(int [],int);
float average(int [],int);
void grade(float);
```

```
//int compute_grade([]);
int n;
int main()
{	int i,total;
	//char g[2];
	int arr[5];
	float avg;
	cout<<"Enter No. of Subject "<<endl;
	cin>>n;
	cout<<"Enter Marks "<<endl;
	for(i=0;i<n;i++)
		cin>>arr[i];
	avg=average(arr,n);
	grade(avg);
	cout<<"\nAverage Marks = "<<avg;
	return 0;
}
float average(int x[], int s)
	{
		int i,sum=0;
		for(i=0;i<s;i++)
			sum=sum+x[i];
		cout<<"\nTotal Value "<<sum;
		float p=sum/s;
		return (p);
	}
void grade(float t)
{
	if (t > 80)
		cout<<"\nA+";
	else if (t > 60)
		cout<<"\nA";
		else if(t > 50)
			cout<<"\nB";
				else if(t > 40)
					cout<<"\nC";
				else
					cout<<"\nF";
}
```

Benefits of Function

Easy to read
Avoid rewriting of the same code
Easy to debug
Better memory utilization

Better Memory utilization
When any program run then first of all main function retrieve the memory
And when main function call to any other function then this other function also come into RAM it means no any function does not come into RAM until it call by other function so this is also saving memory

And if whole program comes into RAM then it might be shortage of memory while if only one function comes into RAM and other function are not in RAM therefore this is less consumption of memory

If any function completes their work then a memory is allocated in RAM which will be release do this is called Better Memory Utilization

The Main Advantages of Function is save the memory suppose

Function is time Consuming Time 3:38
However every time a function is called it takes lot of extra time in executing a series of instruction for task such as jumping to the function
When first Function call to other function then first function takes some time to preserve it because that other function come into RAM then execute it and then it'll release it and first function again reload and then resume it and again function run it then release it
This overhead applies that as much as function call it will applicable

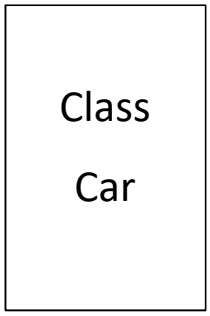
OOPS
OOP Stands for object oriented programming

Oops is a programming approach that are on class and object which contains the data and code that manipulate data.

Procedural programming is about writing procedure or function that

perform operation on the data

Class and Object



Example

Person				
Attribute	**Name**	Ankit	Amit	Aman
	Age	25	24	20
	Location	Varanasi	Delhi	Kanpur

Here Age Name and Address are common are common information which we have three object in a One Class

Class is a user defined data type or blueprint that wrapped data and function in a single entity

Syntax :

class class_name

{

Data + Function

};

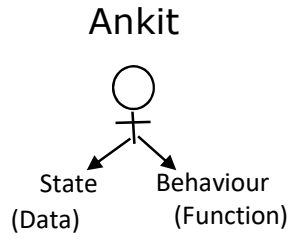
Q. What is object Full explanation

Object is concrete representation of Blue print that is defines by class.

Object have two types of Property

1-**State** -> Data -> Name Age Location

2- **Behaviour** -> Access the data through Function i.e, behaviour



So a class is a template for object and an object is an instance of a class

When the individual objects are created they inherit all the variable and function and function from class

Class is a blueprint of an Object

Class is a description of Object’s property set and set of operation

Class is a mean to achieve encapsulation

Suppose I want to make a group some variable and function related student and I created another group related that student so I have created two class and both group are keeping the information of same student so This is not encapsulation

Encapsulation

When you create all the property of and all function in a single group when you create making this class then it is called Encapsulation

Object is a run time entity

Example

```
#include <iostream>
using namespace std;
class person
{

int age;
string name;
string location;
public:
void input() {
cout<<"Enter Your Name ";
cin>>name;
cout<<"Enter Your Age ";
cin>>age;
cout<<"Ent Your Location ";
cin>>location;
}
void show()
{
cout<<"Name "<<name<<endl;
cout<<"Age "<<age<<endl;
cout<<"Location "<<location<<endl;
}

};

int main()
{
person ankush,ankit,aman;
ankush.input();
ankush.show();
ankit.input();
ankit.show();
aman.input();
```

```
aman.show();
return 0;
}
```

There are Four Principles in OOP (AEIP)

Abstraction

Encapsulation

Inheritance

Polymorphism

Encapsulation

Encapsulation means Binding Data and method within a class, providing control over the accessibility and it prevents external code

```
Class person
{   private:
    Data;
    Function()
    {

    }
};
Main()
```

Example :

```
#include <iostream>

using namespace std;

class Encap

{ int a;

public:

string name;

int age=a;

};

main()
```

```
{
Encap E;
E.name="Gaurav";
E.age=24;
cout<<"Name is "<<E.name<<"\nAge is "<<E.age<<endl;
}
```

In above program run because Attributes are Public

If we want to change one attribute from private then declare function

```
#include <iostream>
using namespace std;
class Encap
{ int age;
public:
string name;
void setValue(int a)
{ age=a;
cout<<age;
}
};

int main() {
Encap E;
E.name="Akhil";
cout<<E.name<<endl;
E.setValue(25);
return 0;
}
```

Output :

Akhil
25

Example

```
#include <iostream>
using namespace std;
class Encap
{ int age;
void show(string n)
{
cout<<"My Name is "<<n<<endl;
}
public:
string name;
void showAge(int a)
{ name="Gaurav";
   show(name);
   age=a;
   cout<<"\nAge is = "<<age<<endl;
}
};

int main()
{
  Encap E;
  E.showAge(29);
  E.name="Akhil";
cout<<E.name<<endl;
return 0;
}
```

Output

My name is Gaurav

Age is 29

Akhil

Example:

```
#include <iostream>
using namespace std;
class Encap
{
int age;
string s;
```

```
string show(string n)
{
s=n;
return s;
}
public:
string name;
void setValue(int a)
{
string t=show("Happy");
cout<<"Name is "<<t<<endl;
age=a;
cout<<"Age is "<<age<<endl;
}
};


main()
{
Encap E;
E.setValue(25);
E.name="School";
}
```

Abstraction

Abstraction refers to the act of represent essential features without including the background details of explanation

Abstraction Provides Generalized view of your class or object by providing relevant information

It is the process of hiding working style of an object,

Example

Nokia 1400 -> Calling, SMS

Nokia 2100 -> Calling, SMS, FM Radio, MP Camera

SAMSUNG A -> Calling SMS, FM Radio, MP3, Camera, Recording, Reading Email

Here How to call, sms theses are abstract

Suppose College Application Name Address, DoB, Semester, Percentage etc.,

If same person is in Hospital Then you'll fill the details like name, Address, DoB, Blood Group, Height, Weight

Here Name, Address, DoB are common Information then it should be considered as a Abstract Class

Encapsulation

The wrapping up of the data function into a single unit is known as encapsulation

The insulation of the data from direct access by the program to called data hiding or information hiding

Encapsulation is the process of combining data function into a single unit called class.

In Encapsulation, the data is not access directly it is accessed through the function present inside the class.

Abstraction

Data abstraction is one of the most essential and important features of object-oriented programming in C++.

Abstraction means displaying only essential information and ignoring the details.

Types of Abstraction:

Data abstraction – This type only shows the required information about the data and ignores unnecessary details.

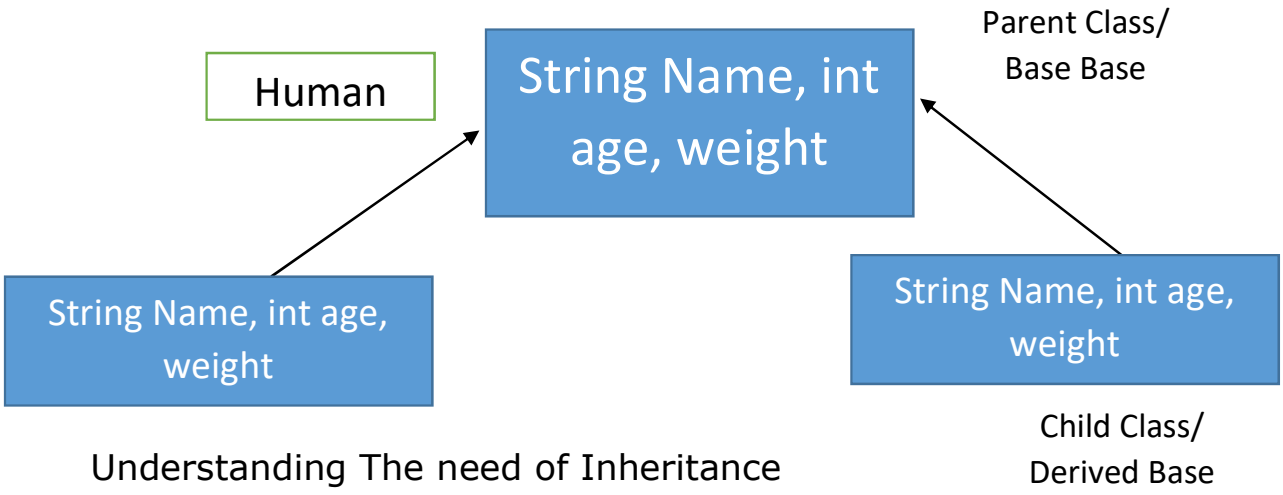
Control Abstraction – This type only shows the required information about the implementation and ignores unnecessary details

```
#include <iostream>
using namespace std;
class car
{
```

```
public:
bool carEngine=false;

void start()
{
 int x;
cout<<"Enter Press 1 or 2 ";
cin>>x;

if (x==1){
carEngine=true;
cout<<"\nEngine is On";
}
else if (x==2)
{
cout<<"\nEngine is Off";
carEngine=false;
}
else
cout<<"\nInvalid Key";
}

void drive() {

if (carEngine)
cout<<"\nYou r ready to drive ";
else
cout<<"\nCan't Drive ";
}
};

main()
{
car c;
c.start();
c.drive();
}
```

Inheritance

Inheritance is the process by which object of one class acquire the properties of object of another class.



Understanding The need of Inheritance

Class is used to describe property and behaviour of an Object

Property names and Values

Behaviour means action

Lets create a class Car

And Its property : price, fuel type engine colour, capacitynd

Method : setPrice(), setFuelType(), setEngine(), setColur()

Suppose I want to create sports car

And Property : price, fuel type, engine, alarm navigator, safeguard

Method :

setAlarm(), setNavigator(), getAlarm(), getAlarm(), getNavigator

previous work again done for some common method it takes time-consuming

again create class different sports car and use that variable which was not in the car class means alarm, navigator, safeguard

and also method but here again apply both objects

if we want to represent sports car then we'll apply both class object because previous car object and sports car object

but when we represent a sports car then we require 8 variables but here we have created only three common variable and car class have five variables but each class requires one object to represent sports car which does not allow according to encapsulation

encapsulation says all the information of a entity should keep in single object

to solve this create separate car class and sports car class

Car Class and Sports Car Class but sports car link to Car Class

So inheritance is a process of inheriting property and behaviour of existing class into a new class.

Existing class = Old class / Parent Class / Base Class

New Class = Child Class / Derived Class

Syntax :

```
Class Base Class
{
};
class Derived_Class : Visibility_Mode Base Class
{
};
```

Example:

```
Class
{
```

```
};
Class SportsCar:public Car{


};
```

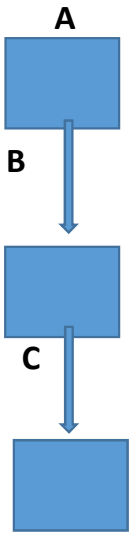
Types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance
- Hierarchy Inheritance
- Hybrid Inheritance

Single Inheritance


```
class A //Parent class
{

};
class B:public A
{

};
```

Multilevel Inheritance

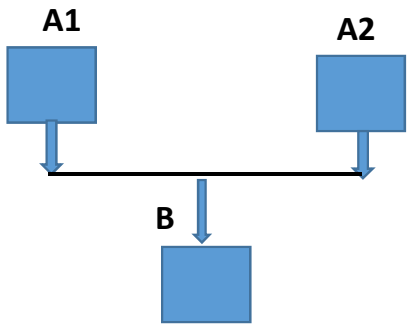



```
class A //Parent class
{
};
class B:public A
{
};
class C:public B
```

{

};

Multiple Inheritance



class A1

{

};

Class A2

{

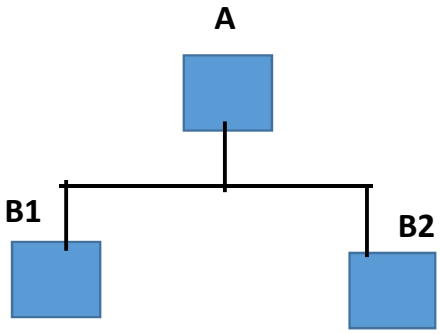
};

Class B:public A1,public A2

{

};

Hierarchy Inheritance



class A

{

};

class B1:public A

{

};

Class B2:public A

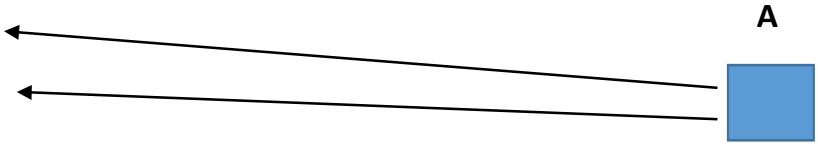
{

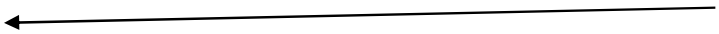
};

Visibility Mode :

Private

Protected



Public 

Example :

```
#include <iostream>
using namespace std;
class A
{
private:
int a;
protected:
          void setValue(int k)
          {
           a=k;
           cout<<"Value is = "<<a<<endl;
          }
};
class B: protected A
{
public:
void setData(int x)
{
 setValue(x);
}
};

main()
{
          B obj;
          obj.setData(20);
}
```

Example

```
#include <iostream>
```

```
using namespace std;
class father {
        protected:
         string sirname="Verma";
};
class son1:father {
        public:
        string name="aman";
        void show()
        {
         cout<<name<<" "<<sirname<<endl;
        }

};

class son2:father{
        public:
        string name="akash";
        void display(){
         cout<<name<<" "<<sirname<<endl;
        }

};

main()
{
        son1 s1;
        son2 s2;
```

```
        s1.show();
        s2.display();
}
```

mixture of inheritance

mixture of inheritance become mixture of more than one inheritance

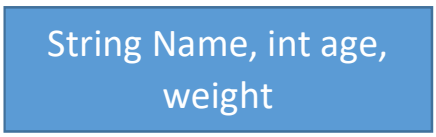
	External Code	Within Class	Derived Class
Public	√	√	√
Protected	X	√	√
Private	X	√	X

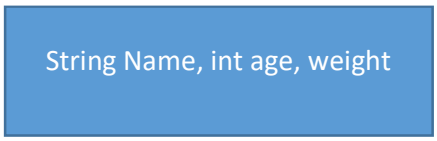
Access Specifier

Within Class
√
√
√

```
#include <iostream>
using namespace std;
class Human
{
private:
```

```
int a;
protected:
int b;
public:
int c;
void fun()
{
   a=10;
   b=24;
   c=41;
cout<<a<<endl<<b<<endl<<c<<endl;
}
};

int main()
{
  Human R;
  R.fun();
return 0;
}
```



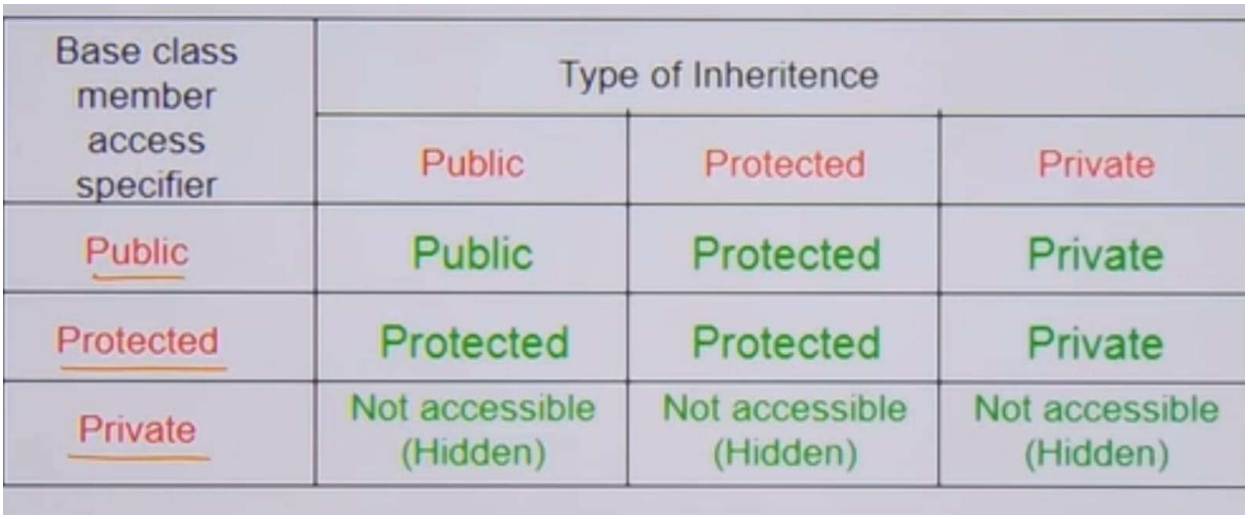


Syntax :

```
class Student:<access Modefier> Human
      Child Class                    Parent Class
```

Parent Class/ Base Class	Child Classef Access Modefier	Inherit-> Result
Public	Public	Public

Public	Protected	Protected
Public	Private	Private
Protected	Public	Protected
Protected	Protected	Protected
Protected	Private	Private
Private	Public	Private



Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

```
#include <iostream>
using namespace std;
class Human
{
private:
string name;
int age,weight;

};

class Student:protected Human
{
```

```
private:
int roll_number,fees;
public:
void fun(string n,int a,int s)
{
name=n;
age=a;
weight=s;
}
void display()
{
cout<<name<<" "<<age<<" "<<" "<<weight<<endl;
}
};

int main()
{
Student A;
A.fun("Rohit",45,49);
A.display();
return 0;
}
```

Error : Private is not accessible

Example

```
#include <iostream>
using namespace std;
class Human
```

```
{
public:
string name;
int age, weight;
};

class Student: private Human
{
private:
int roll_number,fees;
public:
void fun(string n,int a,int s)
{
name=n;
age=a;
weight=s;
}
void display()
{
cout<<name<<" "<<age<<" "<<" "<<weight<<endl;
}
};

int main()
{
Student A;
A.fun("Rohit",45,49);
A.display();
```

```
return 0;
}
```

Output :

Rohit 45 49

Base Class is Public and Child class is private

Public -> Private : private -> goto Public Function and Main()

Example

```
#include <iostream>
using namespace std;
class father
{
private:
int x;
protected:
string sirname="verma";
};
class son1:protected father
{ public:
void show()
 {
cout<<"Aman"<<" "<<sirname<<endl;
}

};
class son2:private father
{ public:
```

```
void disp()
{
cout<<"\nAnkit"<<" "<<sirname;
}
};

main()
{
son1 s1;
son2 s2;
s2.disp();
}
```

Why we do use private suppose in Parent Class private I have variable string religion, fees when I call the function I don’t want to display this two variable we can also create constructor

```
#include <iostream>
using namespace std;
class Human
{
string religion, color;
protected:
string name;
int age,weight;
c};
class Student:private Human
{
private:
```

```
int roll_number,fees;

public:
Student(string x,int y,int weight,int roll_number,int fees)
{
this->name=x;
this->age=y;
this->weight=weight;
this->roll_number=roll_number;
this->fees=fees;
}
void display()
{
cout<<name<<" "<<age<<" "<<weight<<" "<<roll_number<<"
"<<fees;
}
};

int main()
{
Student A("rohit",45,74,101,4500);
A.display();
return 0;
}
```

One name having multiple forms

For example, the operation addition

For two number the operation will generate a sum

If the operands are string then the operation would be produce a third string concatenation

Example 1

```
#include <iostream>

using namespace std;

class sample

{

public:

void show()

{

cout<<"Show the Function ";

}

};


int main()

{

sample obj;

obj.show();

return 0;

}
```

Polymorphism

There are Two Type :

Compile time Polymorphism

Run Time Polymorphism

Compile time has Two Type :

1. Function Overloading
2. Operator Overloading

Run time has One Type :

Virtual Function ->method overriding

Compile time is called when we execute and compiler checks all the commas, semicolon header file name if there is missing then it occurs error

Run time: when we run code and we compile and it compiler checks the code and after this it converts into the code of machine executable ciode means it convert into binary form it is called Run time

```
#include <iostream>

using namespace std;

class Area

{

public:

int calculateArea(int r) // Area of circle

{
return 3.14*r*r;
}
int calculateArea(int l,int b) // Area of rectangle
      {

         return l*b;

       }

};

main()

{

Area A1;
```

```
cout<<A1.calculateArea(4)<<endl;
cout<<A1.calculateArea(5,8)<<endl;
cout<<A1.calculateArea("grv")<<endl; //compile time error and Function is overloading also because same function name is calling
}
```

Operator Overloading

```
#include <iostream>
using namespace std;

class check
{
public:
string addition(string a,string b)
{
return a+" "+b;
}
int addition(int m,int n)
{
return m+n;
}
};
main()
{
check ch;
//ch.addition("aman","singh");
cout<<ch.addition("aman","singh")<<endl;
cout<<ch.addition(15,41);
```

}

Now Suppose we have two objects and I want to apply operator overloading on both object

Lets take complex number : a+ib and add it

3+i2

4+i6

3,4 are real number and i2 is imaginary number lets take two variable real and imaginary

When we trying to add object that is user defined type data

3+i2

4+i6

7+i8

```
#include <iostream>
using namespace std;
class complex
{
int real,img;
public:
complex() // default constructor
{
};
complex(int real, int img)
  {
```

```
    this->real=real;
    this->img=img;
   };
void display()
{over
cout<<real<<" +i"<<img;
}
complex operator+(complex &c)
{
complex ans;
ans.real=real+c.real;
ans.img=img+c.img;
return ans;
}

};
main()
{
complex c1(3,2);
complex c2(4,6);
complex c3=c1+c2;
c3.display();
}
```

Run time has One Type :

Virtual Function ->method overriding

Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism

Derive them using virtual Keyword

Virtual Function are accessible using Object Pointer

Virtual call resolving is done at run time

```
#include <iostream>
using namespace std;
class Animal
{ public:
void speak()
  {
  cout<<"Sound is coming ";
  }
};
class dog:public Animal
{
void speak()
{
cout<<"Barking ";
}
};

main()
{
Animal *p; //p is pointer var. and pointing Animal Type value
p=new dog(); //object created
p->speak();
}
```

Output :

Sound is coming

But I want Barking then use virtual keyword in base class function

Animal *p

p=new dog() // p will decide the which function is called it is storing the address of doc type

```
#include <iostream>
using namespace std;
class Animal
{ public:
  virtual void speak()
  {
  cout<<"Sound is coming ";
  }
};
class dog:public Animal
{
void speak()
{
cout<<"Barking ";
}
};

main()
{
Animal *p; //p is pointer var. and pointing Animal Type value
p=new dog(); //object created
p->speak();
}
```

Q. How can Object-Oriented Design (OOD) improve software maintainability?
OOD improves software maintainability through:

- **Modularity**: By dividing the system into discrete classes, each responsible for specific functionality, it becomes easier to understand, modify, and extend the system.

- **Encapsulation**: By restricting access to the internal state of objects and exposing only necessary interfaces, it reduces the risk of unintended interference and makes the system more robust.
- **Reuse**: Inheritance and polymorphism promote code reuse, reducing redundancy and making it easier to update and maintain the system.
- **Clear Design**:
- action allows designers to focus on high-level design, making the system easier to understand and reason about.

Intention of object oriented modeling and design is to learn how to apply object - oriented concepts to all the stages of the software development life cycle. Object-oriented modeling and design is a way of thinking about problems using models organized around real world concepts. The fundamental construct is the object, which combines both data structure and behaviour.

Types of Models:
There are 3 types of models in the object oriented modeling and design are: Class Model, State Model, and Interaction Model. These are explained as following below.

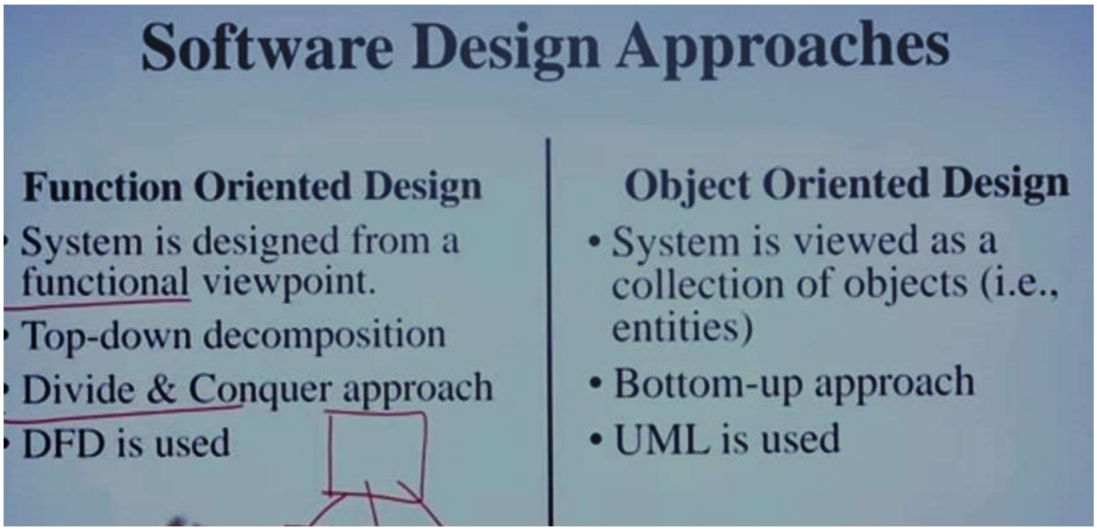
1. **Class Model:**
   The class model shows all the classes present in the system. The class model shows the attributes and the behavior associated with the objects.
   The class diagram is used to show the class model.The class diagram shows the class name followed by the attributes followed by the functions or the methods that are associated with the object of the class. Goal in constructing class model is to capture those concepts from the real world that are important to an application.

2. **State Model:**
   State model describes those aspects of objects concerned with time and the sequencing of operations – events that mark changes, states that define the context for events, and the organization of events and states. Actions and events in a state diagram become operations on objects in the class model. State diagram describes the state model.

3. **Interaction Model:**
   Interaction model is used to show the various interactions between objects, how the objects collaborate to achieve the behavior of the system as a whole. The following diagrams are used to show the interaction model:
   - Use Case Diagram
   - Sequence Diagram
   - Activity Diagram

Purpose of Models:

1. Testing a physical entity before building it
2. Communication with customers
3. Visualization
4. Reduction of complexity



Q What is the role of UML in Object-Oriented Design?
Unified Modelling Language (UML) plays a critical role in Object-Oriented Design by providing a standardized way to visualize the design of a system. UML includes various types of diagrams, each serving different purposes

MODEL

Software modeling involves creating abstract representations of the most significant aspects of the software under study. It provides a means for software engineers, the development team and other stakeholders to analyze, reason about, and understand key elements

of the structure, behaviour, intended use, and assembly considerations of the software. Modeling facilitates making important decisions about the software or components

Modelling is important in object-oriented analysis and design (OOAD) because it helps us understand and develop systems in several ways:

- **Visualization:** Models help us visualize a system as it is or as we want it to be.
- **Structure and behaviour:** Models allow us to specify the structure or behaviour of a system.
- **System construction:** Models provide a template that guides us in constructing a system.
- **Decision documentation:** Models document the decisions we have made.
- **System simplification:** Models can simplify systems at different abstraction levels
- **It Helps the system part by part UML uses in different model .**

Some principles of modelling include:

- Choosing appropriate models
- Expressing models at different abstraction levels for different stakeholders
- Ensuring models are grounded in reality
- Using multiple complementary models to solve complex systems

Modelling is a proven engineering technique important in other fields, such as education, research, and decision-making processes.

Importance & Principles of Modeling from

First we should identify the problem to create a model if we have selected right model then next problem will be solved

Modelling is a proven and well accepted engineering technique, in building architecture we develop architecture models of houses & high rises to help visualize the final product

In Unified Modelling Language a model may be structural emphasizing the organization of the system o

1- Choice of your Model

Right Model will highlight the critical development model, The choice of what model to create has a profound influence on how a problem attacked and how a solution is shaped we need to choose your model well

Wrong model will mislead you and gives the irrelevant data, if it is correct then we can use different type of diagram for different phase in software development

2- Every model may be expressed at different types of precession

Suppose I am an architecture and I have to build a tower building then I have to focus at high level and also ground level of Building tower

Then we need precision for both model High and Ground Level

For example, if you are building a high rise, sometime you need 1000-foot view for instance

3- Best models are connected to reality

All model simplify reality and a good model reflects important key characteristics

4- No single model is sufficient

Suppose if we want to build a building architecture then we need also different building model like electrical model, plumbing model, because no single model is sufficient for a plan
So here we need

UML Unified Modeling Language

We use UML for modeling which is a standard language for specifying and visualizing the artefact of a software system.

the It is industry Standard Graphical Language for specifying Visualizing, Constructing and Document

UML was created by the Object Management Group and UML 1.0 specification draft was proposed to the OMG (Object Management Group) in January 1997

UML is different from the other programming language such as Java C++ COBOL

UML is a Pictorial Language makes a software blueprint

UML can be described as general-purpose visual modeling language to visualize specify construct.

UML is not dependent on any one language or technology

Although UML is generally used to model software system

UML is used to simplify in terms of pictorial model for the software design process

UML Diagram are not only for developers but also require the business users, common people and anybody interested to understand the system

Conceptual Model in UML

As UML describes the real-time system it is very important to make a conceptual model and then proceed **gradually the conceptual model of UML can be mastered by learning the following three major element**

1. **UML Basic Building Blocks**
2. **Rules To connect the Building Block**
3. **Common Mechanism**

Building Blocks of UML Three-Type

- **Three**
- **Relationship**
- **Diagram**

Things are the most important building blocks of UML. Things can be –

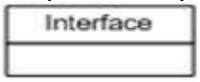
- **Structural**

- **Behavioral**
- **Grouping**
- **Annotational**
- **Structural things** define the static part of the model. They represent the physical and conceptual elements. Following are the brief descriptions of the structural things.

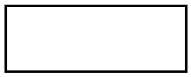
- **Class** – Class represents a set of objects having similar responsibilities.

• Class
• Attributes
• Operation

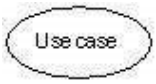
- **Interface** – **Interface defines a set of operations**, which specify the responsibility of a class.



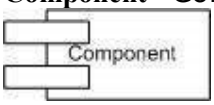
- **Collaboration** –Collaboration **defines an interaction** between elements.



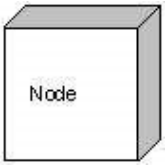
- **Use case** –Use case represents a **set of actions performed by a system** for a specific goal.



- **Component** –Component describes **the physical part of a system.**



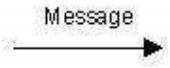
- **Node** – A node can be defined as a **physical element that exists at run time.**



Behavioral Things

A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things –

Interaction – Interaction is a **behaviour consisting of a group of messages** exchanged among elements to accomplish a specific task.



State machine – State machine is useful when the state of an object in its life cycle is important. **It defines the sequence of states an object goes through in response to events.** Events are external factors responsible for state change

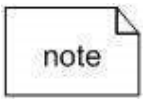
Grouping Things

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available –

Package – Package is the **only one grouping thing available for gathering structural and behavioural things.**

Annotational Things

Annotational things can be defined as a mechanism to **capture remarks, descriptions, and comments of UML model elements.** **Note** - It is the only one Annotational thing available. A note is used to render comments, constraints, etc. of an UML element.



Relationship

Relationship is another most important building block of UML. It shows how the elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

UML relationships

In UML class diagrams, there are several types of relationships that can be used to show how classes, interfaces, and objects are related to each other.

1. **Association**

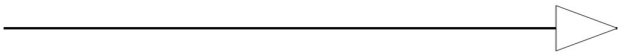
Association is a relationship in which one class has a reference to another class.

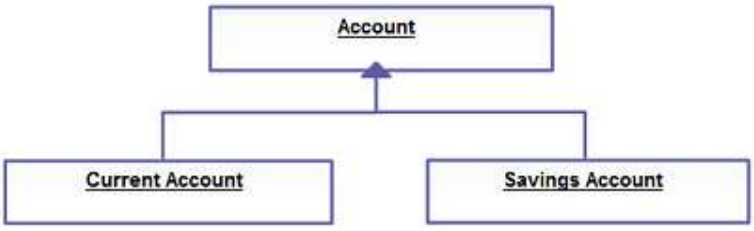
This is represented by a line connecting the two classes, ssociation is basically a set of links that connects the elements of a UML model. It also describes how many objects are taking part in that relationship.

directionality. Association is used to model a "uses-a" relationship, where the source class is dependent on the target class.

2. Inheritance (Generalization)

Inheritance **is a** relationship in which a derived class inherits the properties and methods of another class (**base class**). This is represented by a solid line with an arrow pointing from the derived class to the base class. Inheritance is used to form a "is-a" relationship.

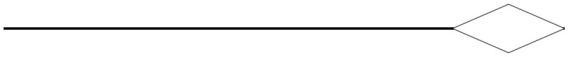


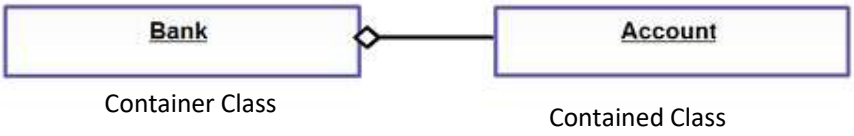


3. Aggregation

Aggregation is a relationship in which one class (the aggregate class) contains one or more objects of another class (the part class). This relationship is represented by a diamond shape on the aggregate class, and a line connecting the aggregate class to the parts. The diamond shape indicates that the aggregate class has ownership of

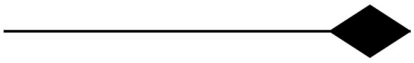
the parts. Aggregation is used to model a "**has-a**" relationship, where the aggregate class is composed of the parts.

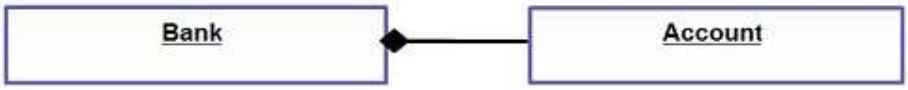




4. Composition

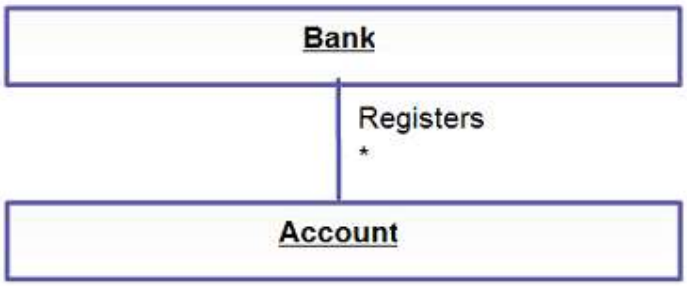
A composition is a strong relationship between two classes where an instance of one class cannot exist without an instance of the other class. Composition is represented by a filled diamond shape on the container class and a solid line connecting it to the contained class.



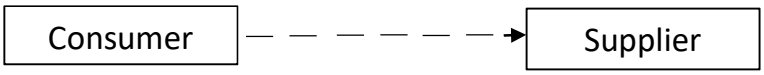


5. Multiplicity

An example of this kind of association is many accounts being registered by the bank; hence, the relationship shows a star sign near the account class (one to many and many to many etc). When it comes to class diagram relationships, this is one of the most misunderstood relationships.

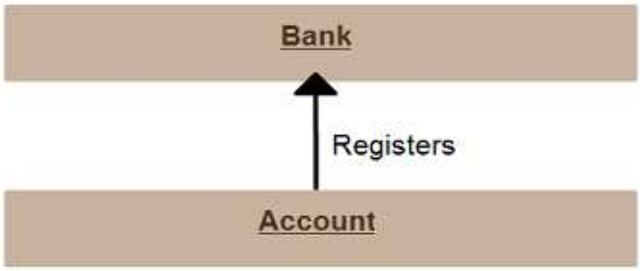


6. Dependency Relationship



7. Directed Association

By default, an association that exists between classes is bi-directional. Ideally, you may illustrate the flow of the association by utilizing a directed association. The arrowhead indicates the container-contained relationship.



Three important types of UML Modeling

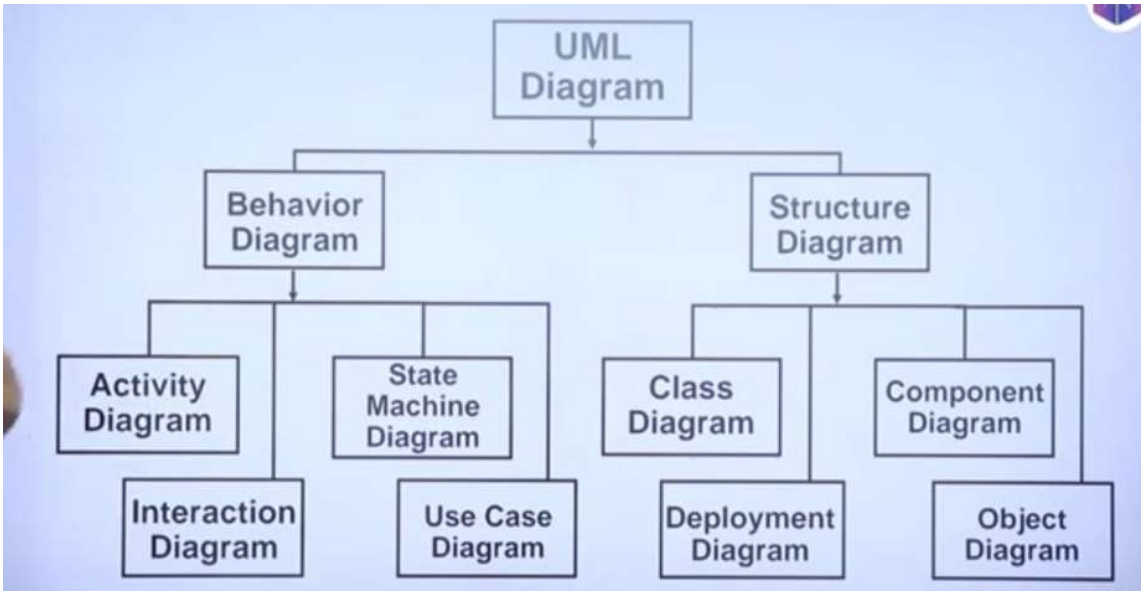
1. **Structural Modeling**
2. **Behaviour Modeling**

The role of Object Model in design

In the context of design, the importance of the object model is that provides a semantic foundation for UML's design notation The meaning of many features in UML can be understood by interpreting them as statement about sets of connected intercommunication objects.

UML diagram can be drawn to represent particular run-time configuration of objects.

https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_implementation_strategies.htm



Class and Object

The data and Functionality in an object oriented system in distributed among the objects that exist while the system is running. In general, there will be lots of **part object** each describing part and so strong different data but each having the same structure

The common structure of a set of a object which represent the same kind of entity is described by Class.

The UML notion of a class is similar to that programming language such as C++ and Java

```
public class Part
{
private:
string name;
long number;
double const;
public:
  part(string nm,long num,double cst) {
  name=nm;
  number=num;
  const=cst
}
```

```
public:
string getName()
{
return name;
}
string getNum()
{
return name;
}
double getConst()
{
return const;
}
```

In UML classes are represented graphically by rectangular icon divided into **three compartments containing the name, attribute and operation**

Part
-name:string -number:long -cost:double
+Part(nm:String,num:long.cst:double)
+getName():String +getNumber():long
+getConst():double

The Part Class represented in UML

Object Creatrion

Class defined at compile time and object created at run time

An instances of class

Part myScrew = new Part(“screw”,542664,254.238);

The object created by the line of code above could be depicted in UML

myScrew: Part
name=”screw” number=542664 cost=254.238

The object is represented by a rectangle divided into two components the upper component contains the object’s name followed

The purpose of Class Diagram

The purpose of Class Diagram is to model the static view of an application. Class diagram are the only diagram are only diagrams that can be directly mapped with object-oriented language and thus widely usd at the time of construction

Class name
Class Attribute
Class Operation

Student
Name : string Roll No : int
Createrecord() : void

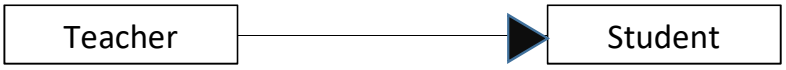
Now add Visibility notation : + public, - private, #protected

Student
+Name : string - Roll No : int
Createrecord() : void

Now add Relationship in classes

Addociation : shows static relationship between the classes

1 1*



1-one, 0..1-> 0 or 1, 0..*->0 or more, 1*-> 1 or more, Exact number (3,4, or 5)

The purpose of Object Diagram

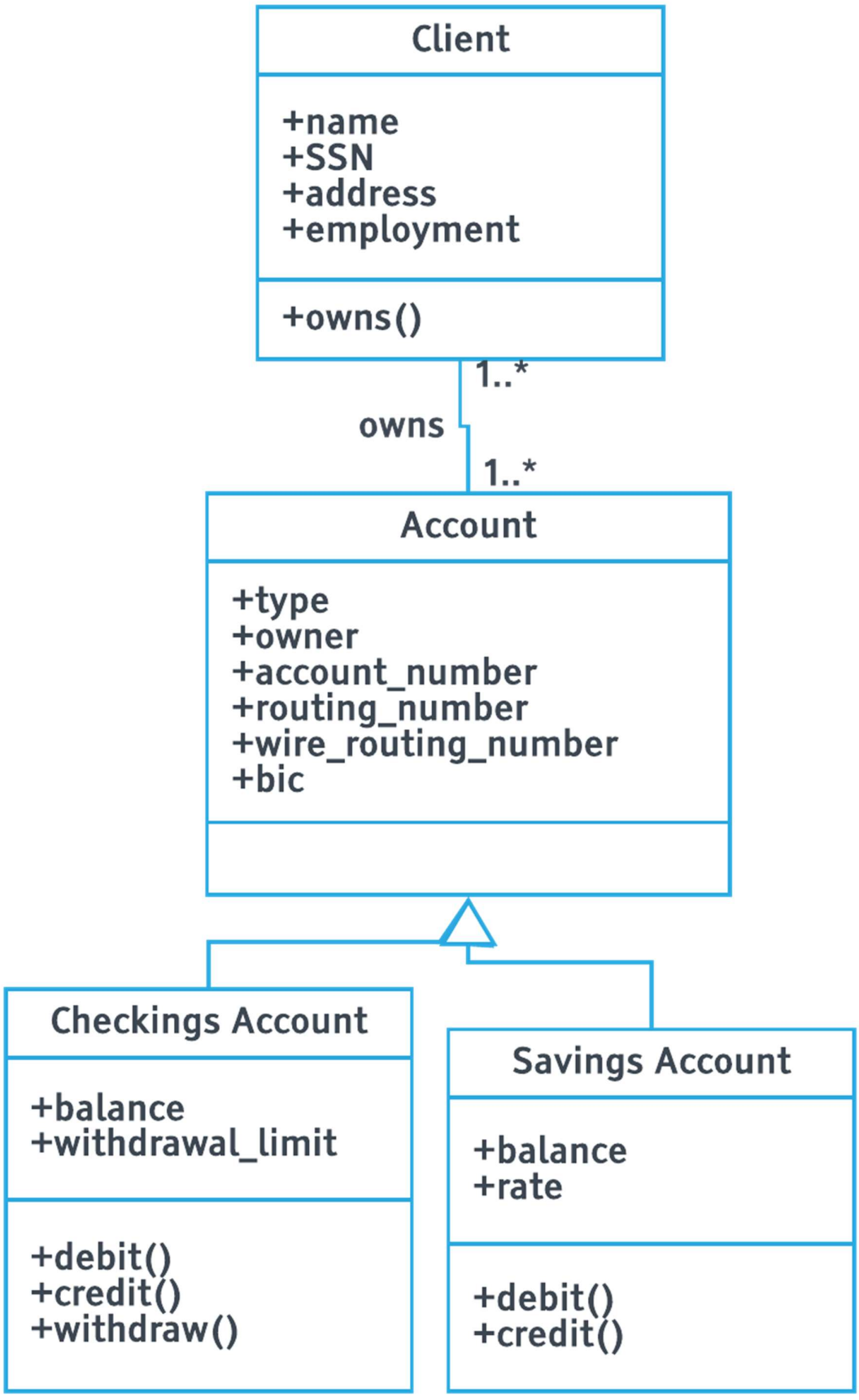
The Purose of Object Diagram should be clearly to implement it practiacally the purpose of Object diagram are similar to class diagrams

The differenec is that a class diagram represent an abstract model consisting of class and their relationship However an object diagram represent an instance at a Particular moment

Everything You Need to Know About UML Class Diagram Relationships

Class Diagram

Class UML diagram is the most common diagram type for software documentation. Since most software being created nowadays is still based on the Object-Oriented Programming paradigm, using class diagrams to document the software turns out to be a common-sense solution.


Client
+name
+SSN
+address
+employment
+owns()
1..*
owns
1..*
Account
+type
+owner
+account_number
+routing_number
+wire_routing_number
+bic
Checkings Account
+balance
+withdrawal_limit
+debit()
+credit()
+withdraw()
Savings Account
+balance
+rate
+debit()
+credit()

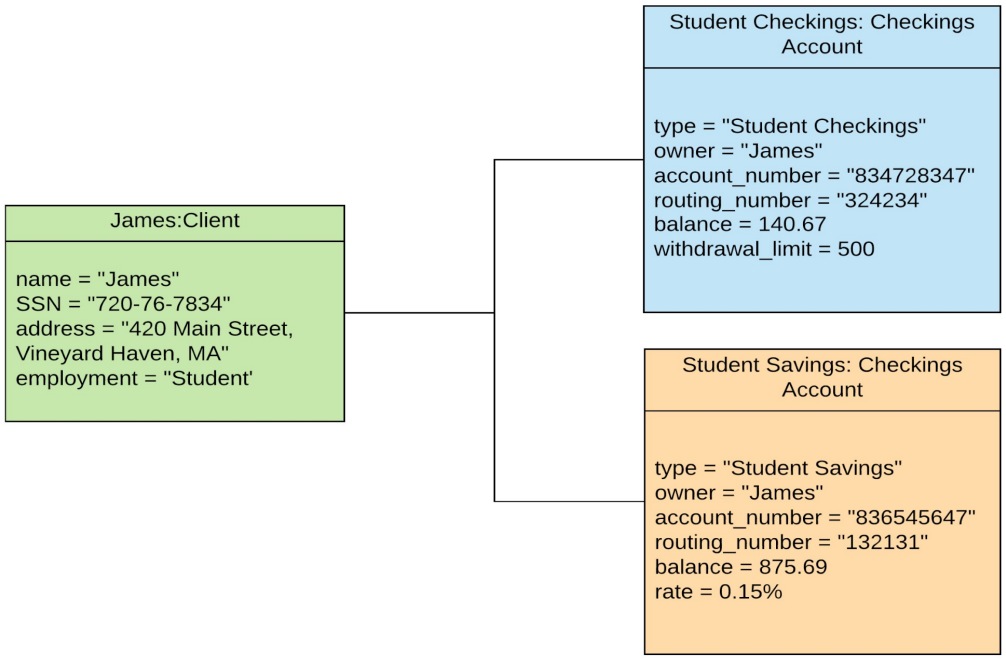
The example above shows a basic class diagram. The ‘Checkings Account’ class and the ‘Savings Account’ class both inherit from the more general class, ‘Account’. The inheritance is shown using the blank-headed arrow. The other class in the diagram is the ‘Client’ class. The diagram is quite self-explanatory and it clearly shows the different classes and how they are interrelated.

Object Diagram

When we discuss structural UML diagrams, we have no choice but to delve deeper into computer science-related concepts. In software development, Classes are considered abstract data types, whereas objects are instances of the abstract class.

Object is an instance of class

Object diagram shows a static view of the system at given particular time instance



Class student {

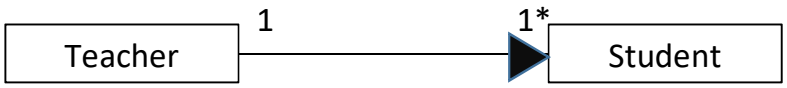
int name;

```
public:
int n,roll_no;
n=name;
}
};
Main ()
```

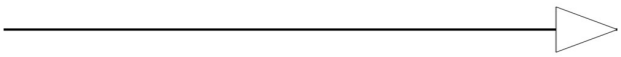
Object Name : Class Name
Attribute

+ S1 : Student
- name=”ashoka”
+ roll_no=101

Addociation : shows static relationship between the classes

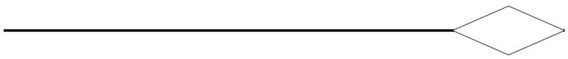


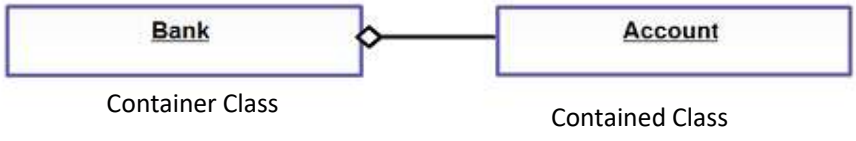
1. **Inheritance (Generalization)**



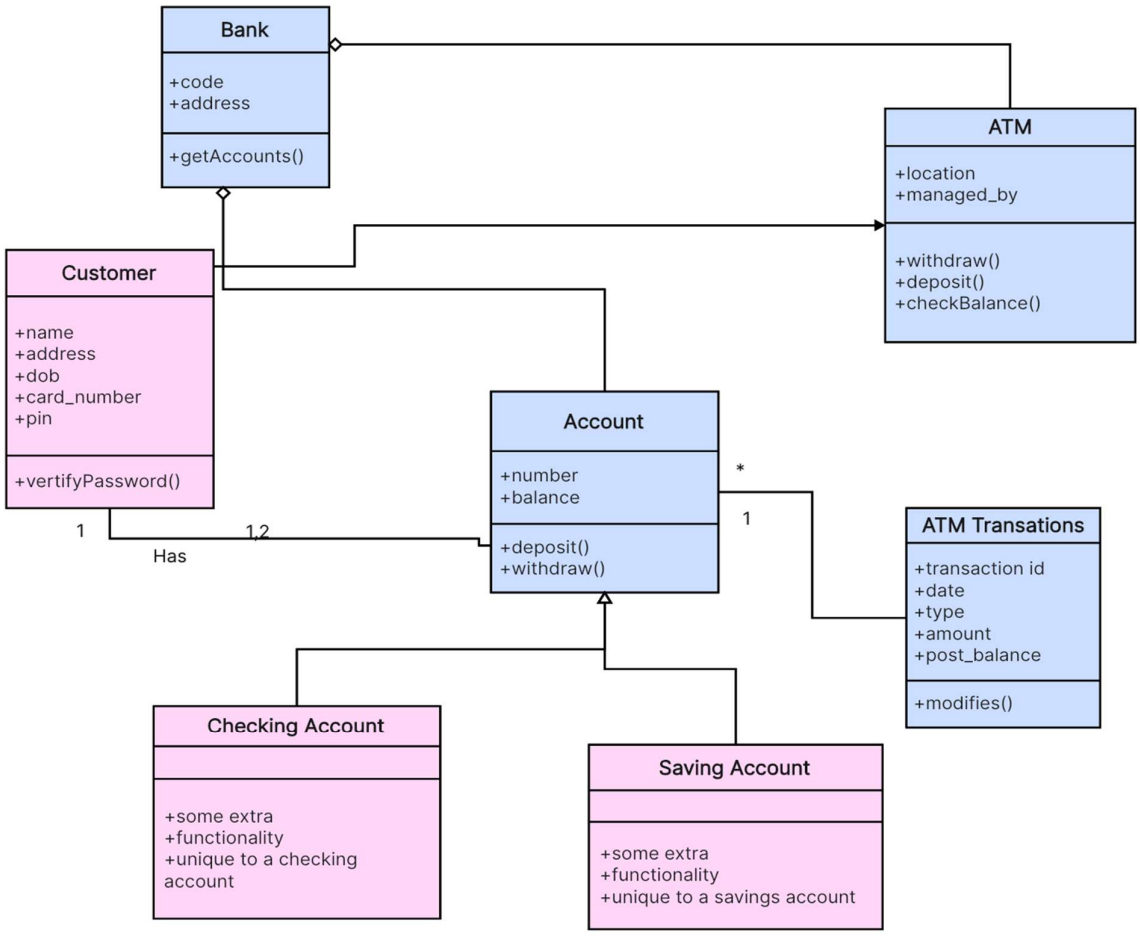
1. **Aggregation**

Aggregation is used to model a "**has-a**" relationship, where the aggregate class is composed of the parts.

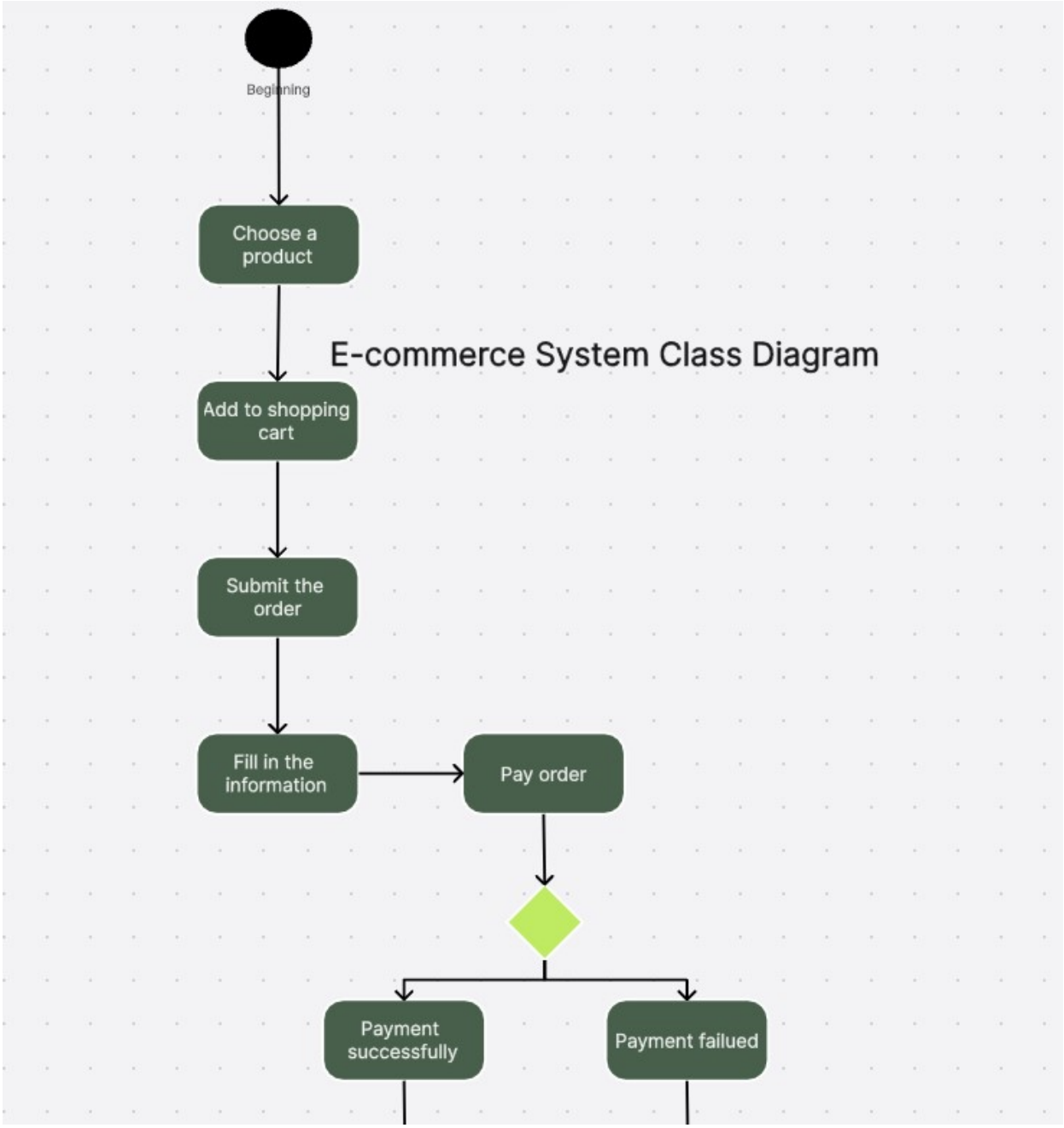




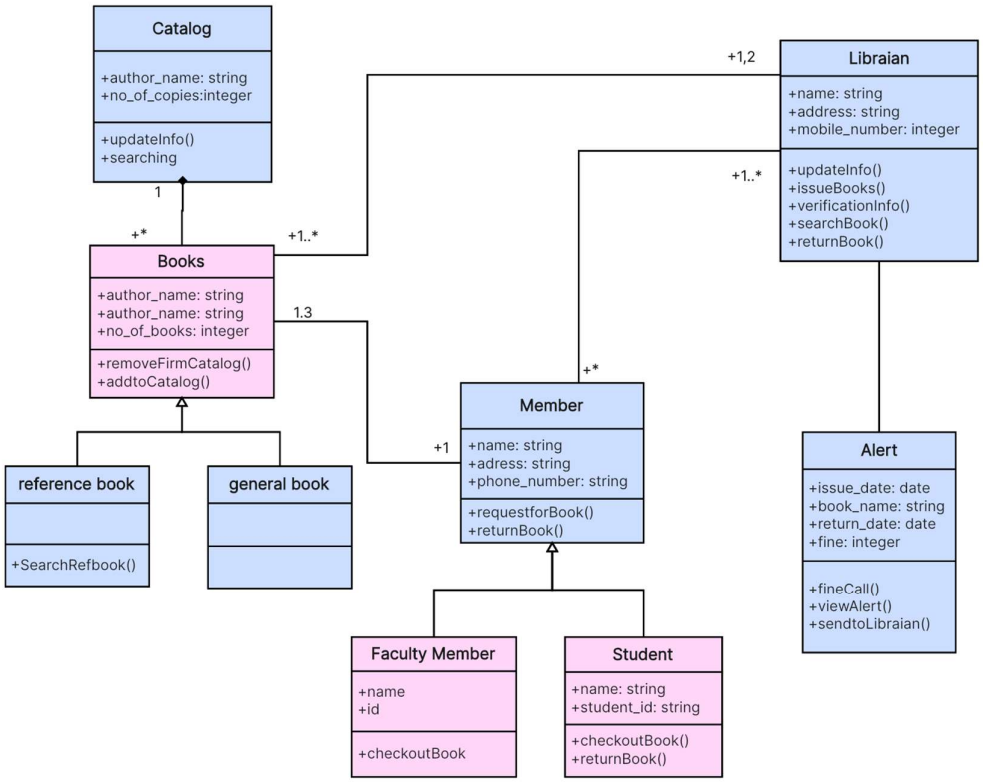
Banking Diagram given Below :

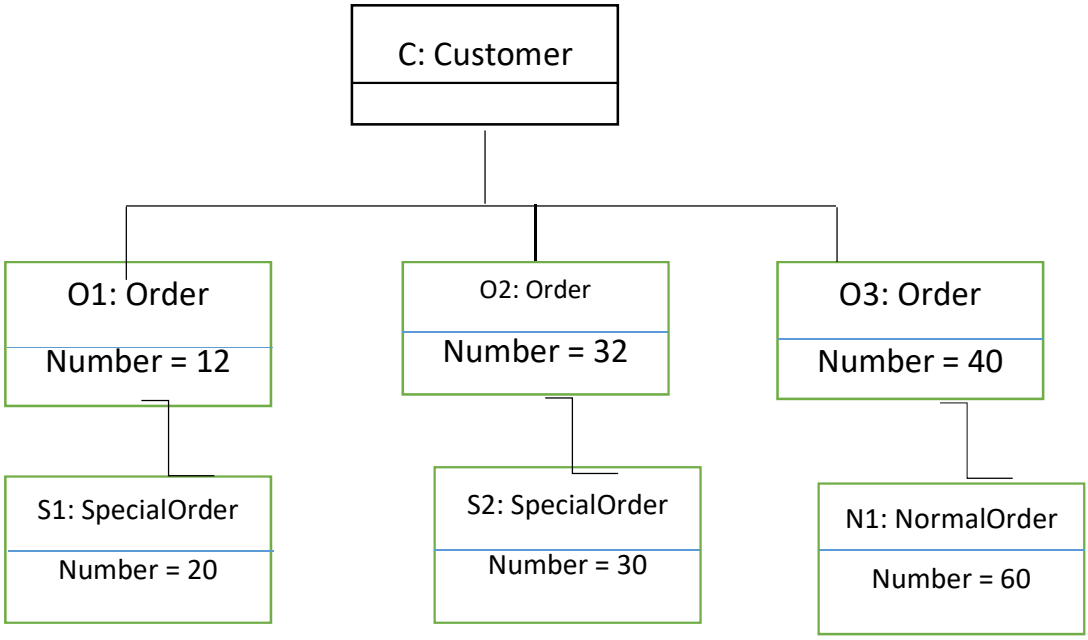


Ecommer Class Diagram


Beginning
Choose a product
E-commerce System Class Diagram
Add to shopping cart
Submit the order
Fill in the information
Pay order
Payment successfully
Payment failued

Library Management





What is a use case model?

A use case model is a visual representation of the interactions **between an actor and a system.** As PMI also notes, use case models depict processes, which helps to further express preconditions and triggers.

A use case model is commonly expressed using UML (Universal Modeling Language). In these visualizations,

Here Use case diagram model the behaviour of system

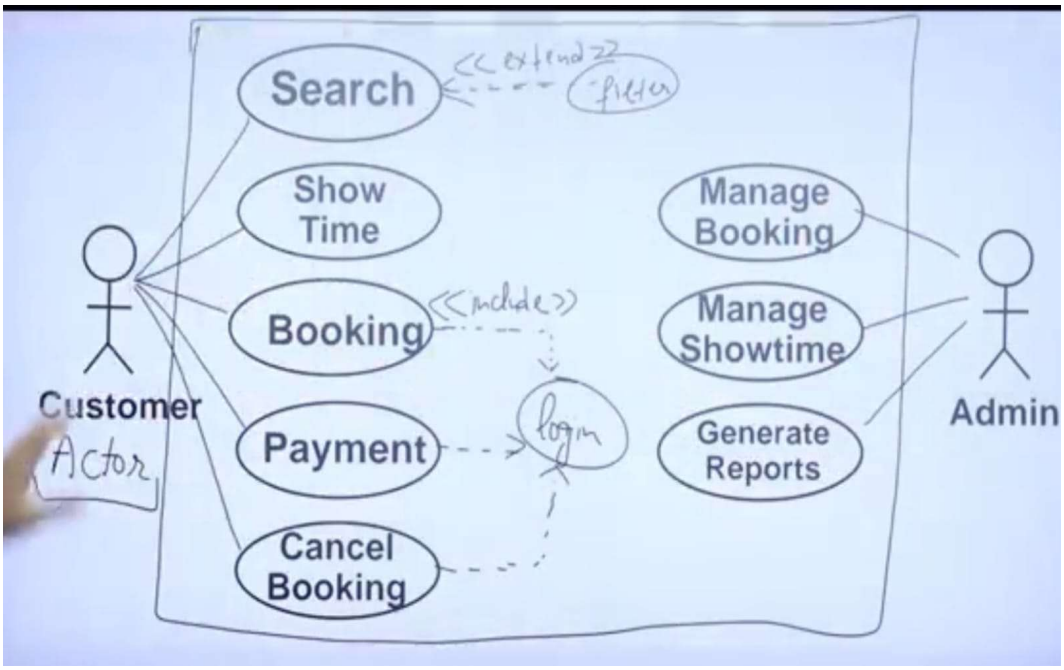
used to illustrate the functional requirement of the system and interaction with external agent

take example of **Bookmyshow.com** first you get **search box** with extra features like any language i.e, shown in <<extend>>

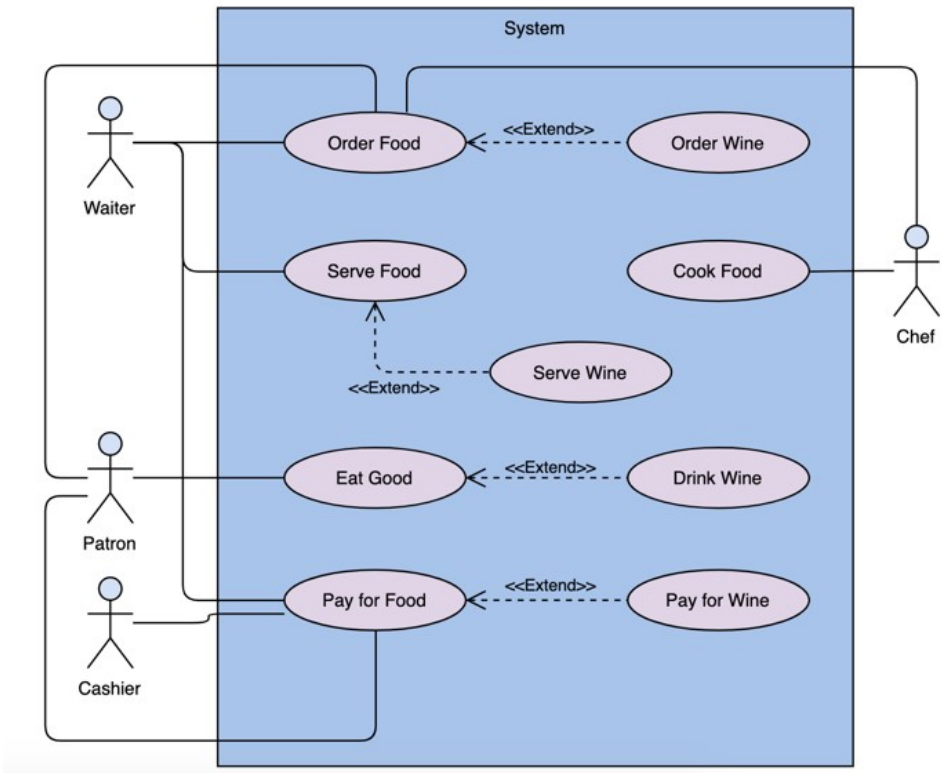
Here we use Actor for customer

Examples

Actors can be **customers, suppliers, patrons, passengers, authorities, or banks**.



Use case model example:



Relationship

Is a Relationship

It two or more classes or entity have relationship i.e, called Association

Banana is a Fruit

Association have some type

- Aggregation
- Composition
- Inheritance

For example: Vehicle and Two_Vehicle

Two_Vehicle is a Vehicle

Person and Student

Student is a Person

So above example incorporates Inheritance with is a keyword

Now Discuss Parent Class and Child Class

Banana is a Fruit: So Fruit is a Parent class and Banana is Child Class

Activity Diagram

The Activity Diagram is a flowchart to **represent the flow of control among the activities in a system** it shows all control flow from one operation to another

We've already used usecase, communication sequence class diagram which shows the message flow from one object to another object

But in Activity diagram message flows one activity to another

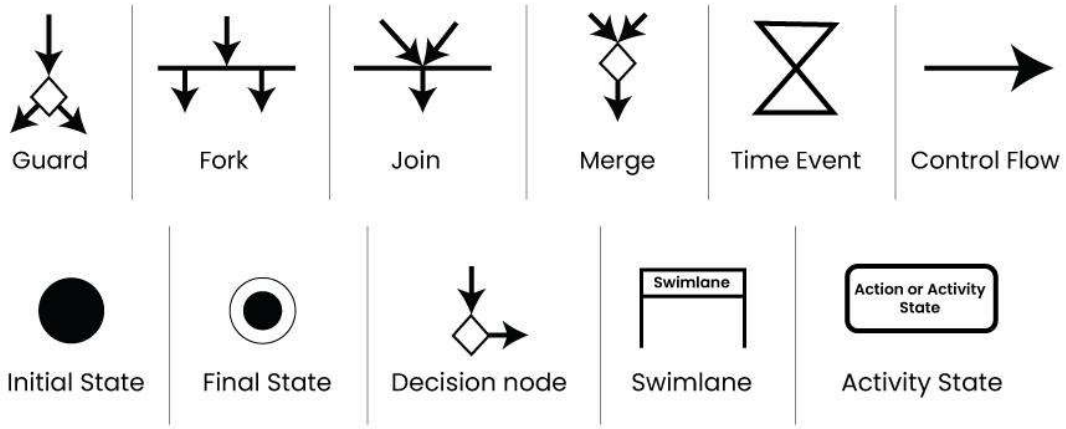
Here Flow of operation can be sequential branched or concurrent

This diagram considered as the flowchart, but they are not

Activity diagrams are an essential part of the Unified Modeling Language (UML) that helps visualize workflows,

processes, or activities within a system.

Activity diagrams show the steps involved in how a system works, helping us understand the flow of control. They display the order in which activities happen and whether they occur one after the other (sequential) or at the same time (concurrent).



1. Initial State

The starting state before an activity takes place is depicted using the initial state.

2. Action or Activity State

An activity represents execution of an action on objects or by objects

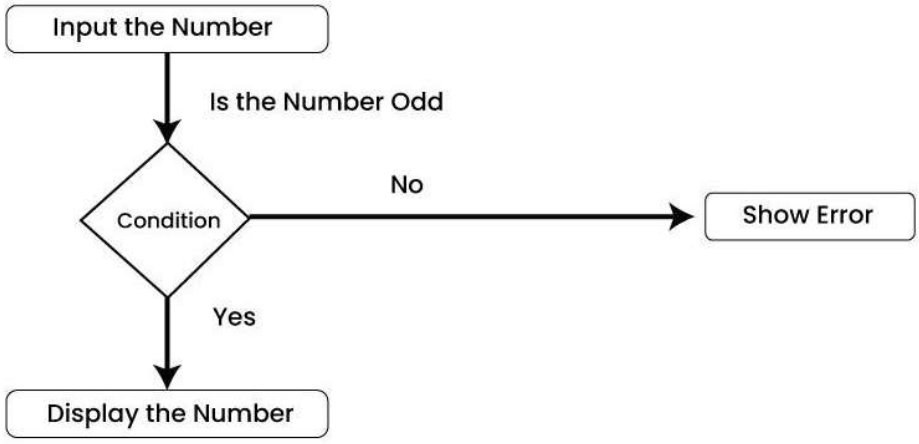
3. Action Flow or Control flows

Action flows or Control flows are also **referred to as paths and edges**. They are used **to show the transition from one activity state to another activity state.**

4. Decision node and Branching

When we need to make a decision before deciding the flow of control, we use the decision node. The outgoing arrows from the decision node can be labelled with conditions or guard expressions

An Activity Diagram using Decision Node

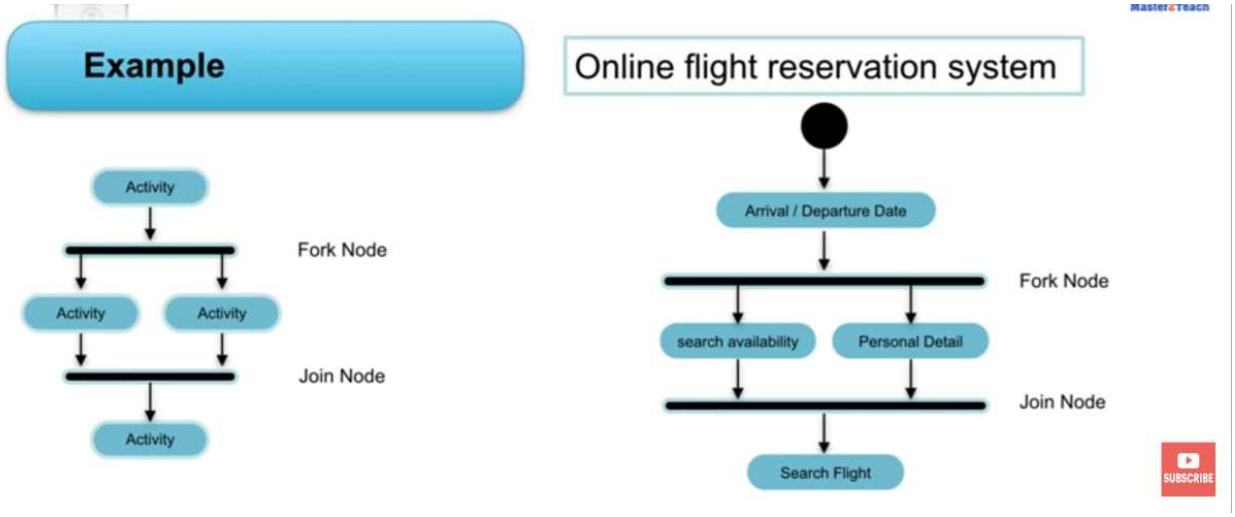


5. Guard

A Guard refers to a statement written next to a decision node on an arrow sometimes within square brackets.

6 Fork

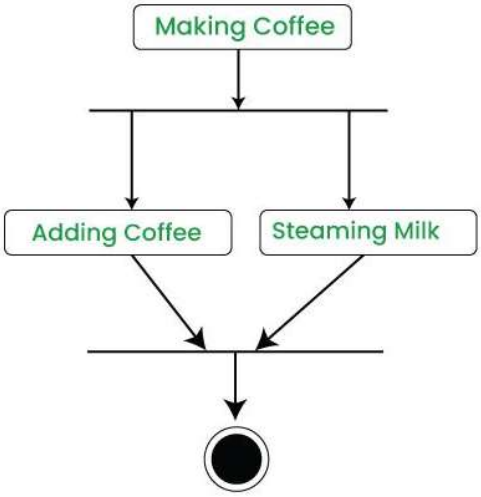
A fork shows splits a single activity flow into two concurrent activities which represented as shown in the figure



7. Join

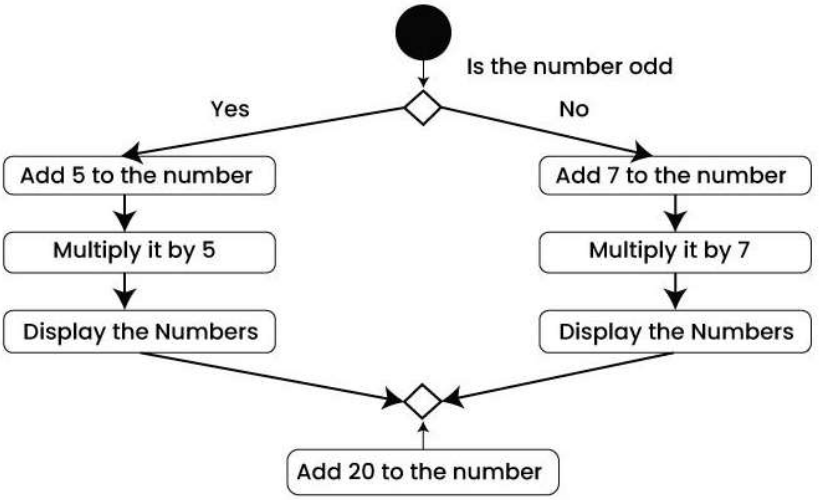
Join nodes are used to support concurrent activities converging into one. For join notations we have two or more incoming edges and one outgoing edge.

When both activities i.e. steaming the milk and adding coffee get completed, we converge them into one final activity.



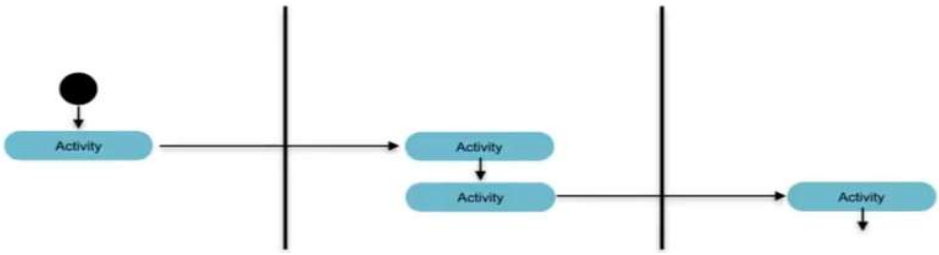
8. Merge or Merge Event

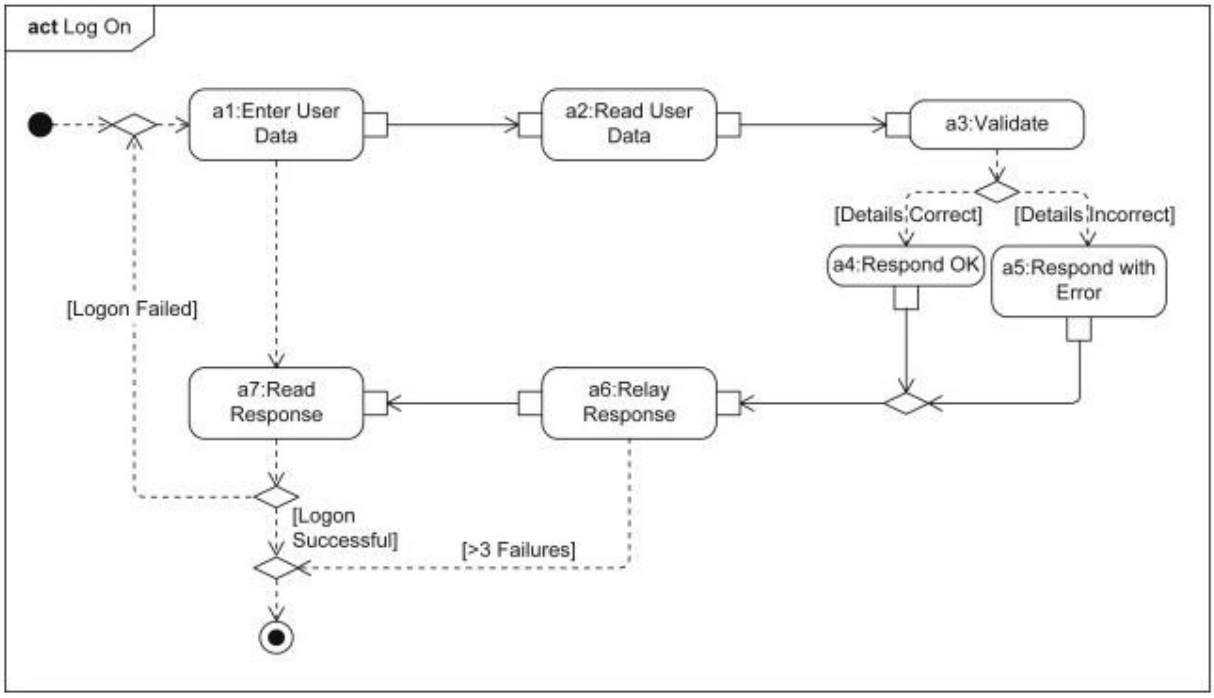
Merge is similar to Join where two activities are merged with condition only one activities flows forward



9. Swimlanes / Partition

We use Swimlanes for grouping related activities in one column. Swimlanes group related activities into one column or one row. Swimlanes can be vertical and horizontal.





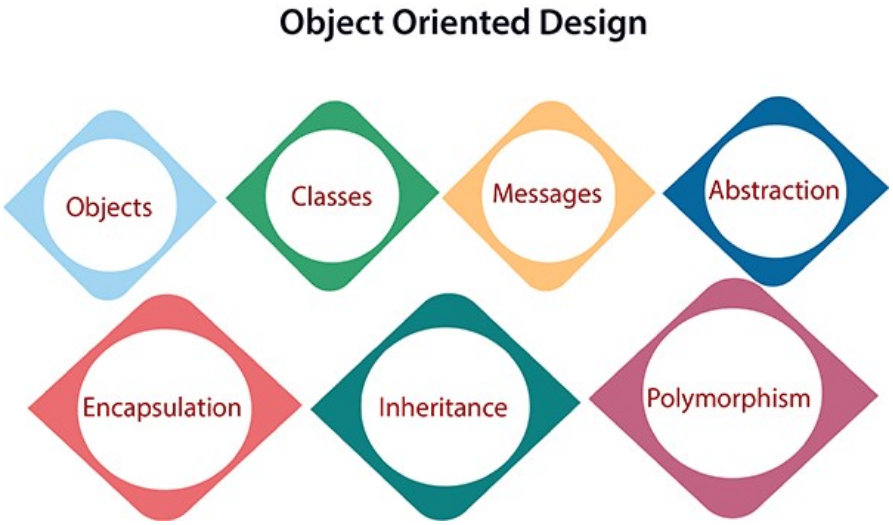
Object-Oriented Design

this is the

In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities). The state is distributed among the objects, and each object handles its state data.

For example, in Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data.

The different terms related to object design are:



Object Oriented Design in Software Engineering is a pictorial representation and this is a software design technique, i.e, used in object oriented programming

1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.
2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.
3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.
4. **Data Abstraction** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.

   Abstraction

5. **Data Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.
6. **Inheritance:** OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.
7. **Polymorphism: OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name.** This is known as polymorphism, which allows a single interface is performing functions for different types. Depending on how the service is invoked, the respective portion of the code gets executed.

Abstraction

Abstraction refers to the act of representing essential features without including bakcground details of explanation
means it is hiding the unwanted data and giving relevant data

Abstraction also focuses what object does

This is outer layout in terms of design of the system

Encapsulation :

The data is wrapped in the class meas it hides code & data into single unit to protect the data from outside the class

where as Encapsulation This is inner layout in terms of design of the system

the above points are programming features

Object Oriented Methodology (OMT)

Object Oriented Analysis developed by **Coad and Yourdan** described step by step and developed object-oriented system model in 1991

Object Oriented Design developed by Grady Booch and He also developed UML

Object Oriented Modeling Technique developed by James Rumbaugh and Lorensen in 1991

Object Oriented software engineering developed by Jacobson

Coad and Yourdan invoves the identification of classes and object

Identification of structures **is a** relationship and **whole part** relationship

Classes

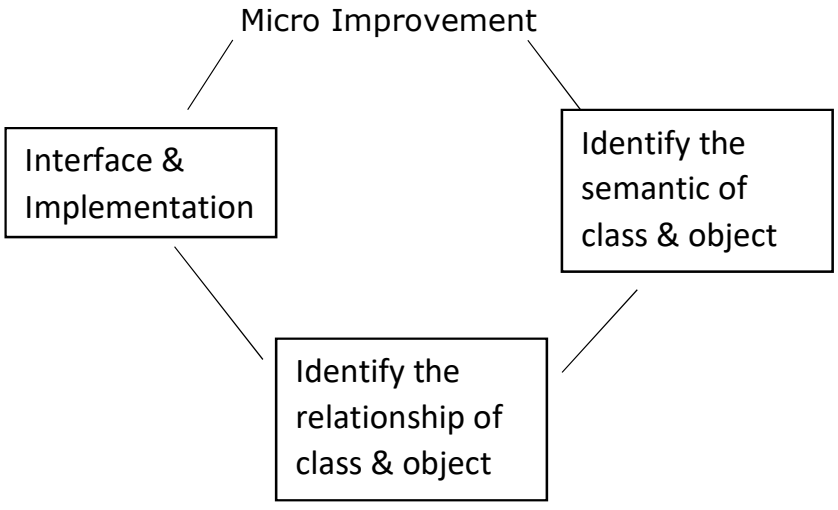
Information of Object of the system

Grady Booch Methodology

Appraches to S/w improvement

Micro Improvement

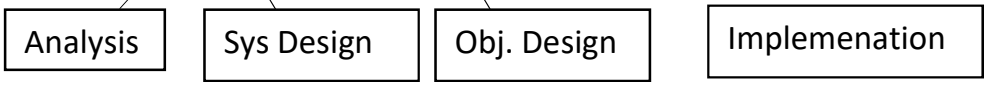
Macro Improvement



Macro development process

- Basic needs of S/w (Conceptualization)
- Analysis
- Design and Development

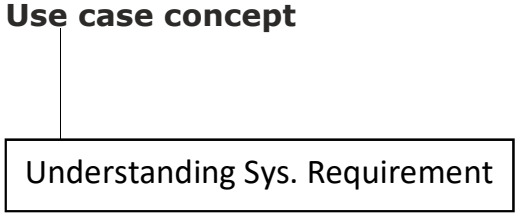
OMT of James Rambaugh



Jacbson Methodology (OOSE)

It covers the entire life cycle and it is Hear of methodology is Use Case Mathodology

Use case concept



Object Oriented Analysis

It determined the system requirement and recognized the classes and relationship between classes

The main purpose of OOA is the recognizing application domain & specific requirement of the system

Combining three techniques of OOA (Object Oriented Analysis)

1. **Object Modeling**
2. **Dynamic Modeling**
3. **Functional Modeling**

Object Modeling : Its instance of class and develops the static structure regarding to the object.

It recognizes the object, relationship between object & Class

The Process of Obj Modeling :

RRUDD

1. Recognize Obj and Grouped into Classes
2. Relationship between the classes
3. User object model diagram is generated
4. Define attribute of User Object
5. Define the Operation need to perform on Classes

Dynamic Modeling

Examin the behaviour of the Object regarding time and external changes

Process of Dynamic Modeling

1. State of every object is recognise
2. recognize the event
3. Generate the dynamic model diagram with state transion diagram
4. communicate with every state diagram
5. verify state transition diagram

Functional Modeling

It shows the process executed in an object and how the data changes when it most between the method

Process of Function Modeling

1. All input outputs are recognized
2. create a data flow diagram to show the functional dependency
3. identify the nature of every function
4. Identify the constraint

Design Analysis Algorithm

Algorithm is the step by step method performing some action

it is step by step solving a problem or doing a task

Common Terms in Design Algorithm :

1) Variable
Specific location in computer memory used to store

2) Data Type
Set of variable takes its values

3) Statement : it’s a line of code

Property of Algorithm

1) **input** : Algorithm uses values from a specifies set
2) **Output** : for each input it specifies produces from a specific task
3) **Precesion** : Steps or steps precisely defined
4) **Correctness** : Input is defined for that output is correct
5) **Fitness** : it produces output after finite number of steps for each input
6) **Determination** :Result should be guaranteed
7) **Generality** : Procedure apply to all problem not a special subset

Algorithm can be express in different notation
Natural Language
Pseudo code
Flowchart
Programming Language

Design Optimization in OOAD

Object-oriented analysis and Design (OOAD) is a crucial phase in software development. In this phase, the system requirements are analyzed and translated into a well-designed object-oriented model. **Design optimization in OOAD focuses on improving the quality**, performance, maintainability, and scalability of the system design.

Principles of Design Optimization

The principles of design optimization emphasize modularity, encapsulation, and low coupling to foster maintainable and scalable software architectures. Here are some principles for design optimization:

Abstraction: Hide unnecessary details and focus on essential aspects of the system, allowing for a simpler and clearer design.

Encapsulation: Bundle data and methods into cohesive units, limiting access to the internal workings of an object.

Modularity: Divide the system into independent modules with well-defined interfaces, promoting reusability and maintainability.

cohesion=एकजुटता

intra – अंदर : cohesion is intramodule

In technical term Modules should display **High Cohesion and Low coupling**
High Cohesion : within Module means grouped in a module Intra Module
suppose we have attendance operation us needed like **mark, view, delete then it is related a single function so put into a single module**
means it is under module different function are grouped and within module all the function are perorming similar operation

Coupling : Coupling within two module here coupling should be low dependency module we have to minimize these dependency and it'll be **low coupling**

Separation of Concerns: Address different aspects of the system independently, such as user interface, data management, and business logic.

Techniques for Design Optimization

Techniques for design optimization include refactoring, applying design patterns, and performance tuning to enhance software quality and efficiency.

Refactoring:
It is restructuring the existing code without altering its external behavior to improve readability, maintainability, and performance.

Design Patterns:
Applying established design solutions to common problems to enhance flexibility, maintainability, and scalability.

Performance Tuning:
Identifying and eliminating bottlenecks through profiling, caching, algorithm optimization, and other performance improvement methods.

Component Reuse: Utilizing existing libraries, frameworks, and components to reduce development time, enhance reliability, and promote consistency.

Design Reviews: Conducting peer reviews and architectural inspections to identify design flaws early in the development process and ensure quality.

Implementation of Control

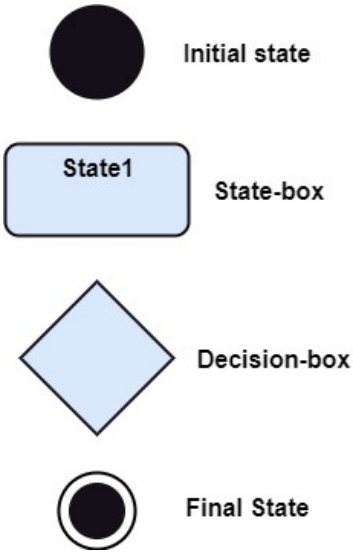
The object designer may incorporate refinements in the strategy of the state–chart model. In system design, a basic strategy for realizing the dynamic model is made.

The approaches for implementation of the dynamic model are −

- **Represent State as a Location within a Program** − This is the traditional procedure-driven approach whereby the location of control defines the program state. A finite state machine can be implemented as a program. A transition forms an input statement, the main control path forms the sequence of instructions, the branches form the conditions, and the backward paths form the loops or iterations.

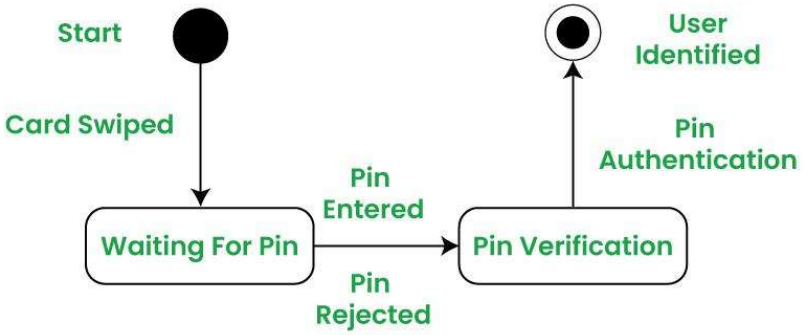
- **State Machine Engine** – This approach directly represents a state machine through a state machine engine class. This class executes the state machine through a set of transitions and actions provided by the application.

- **Control as Concurrent Tasks** – In this approach, an object is implemented as a task in the programming language or the operating system. Here, an event is implemented as an inter-task call. It preserves inherent concurrency of real objects.



Why State Machine Diagram?
A statechart sometimes known as a state machine diagram,
Since it records the dynamic view of a system, **it portrays the behavior of a software application**. During a lifespan, an object underwent several states,

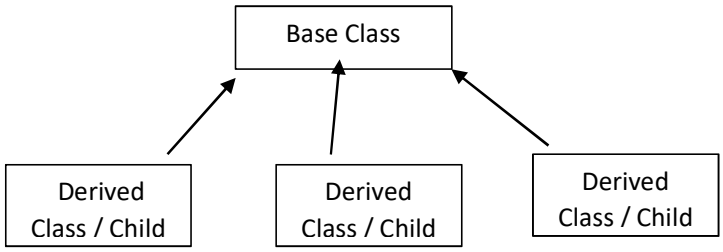
A State Machine Diagram for user verification



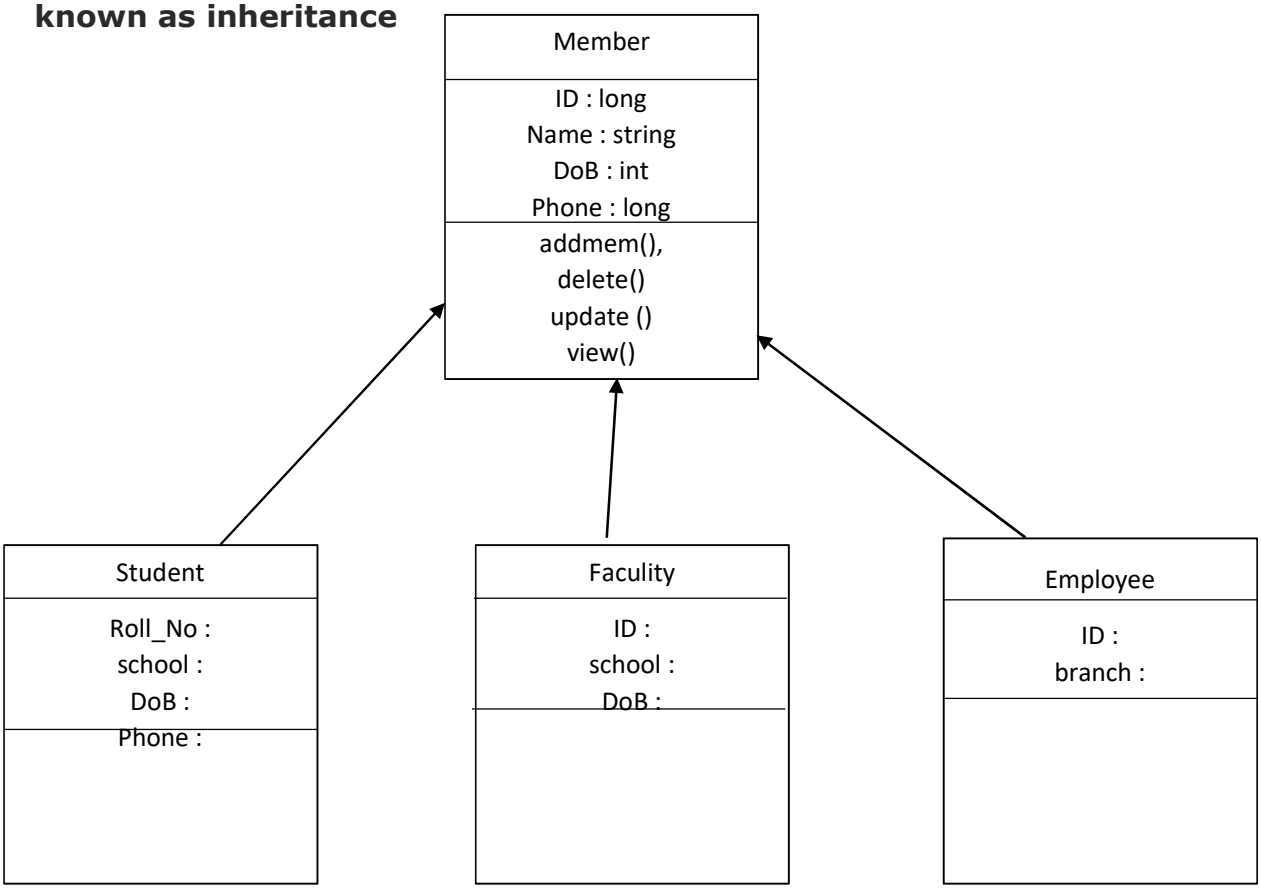
State Machine Diagrams | Unified Modeling Language (UML)

Adjustment of Inheritance
if you know any concept in the study or knowledge i.e, arranged in the hierarchy and means we enter in next level and then next level which type of hierarchy that is Inheritance
we are acquiring the fearures of base class so we may organise the our knowledge we have many classes all classes inheriting to the upper class

child class will aquire the information and inherting the property of parent class.



These derived classes inherit the attribute of base class these process known as inheritance

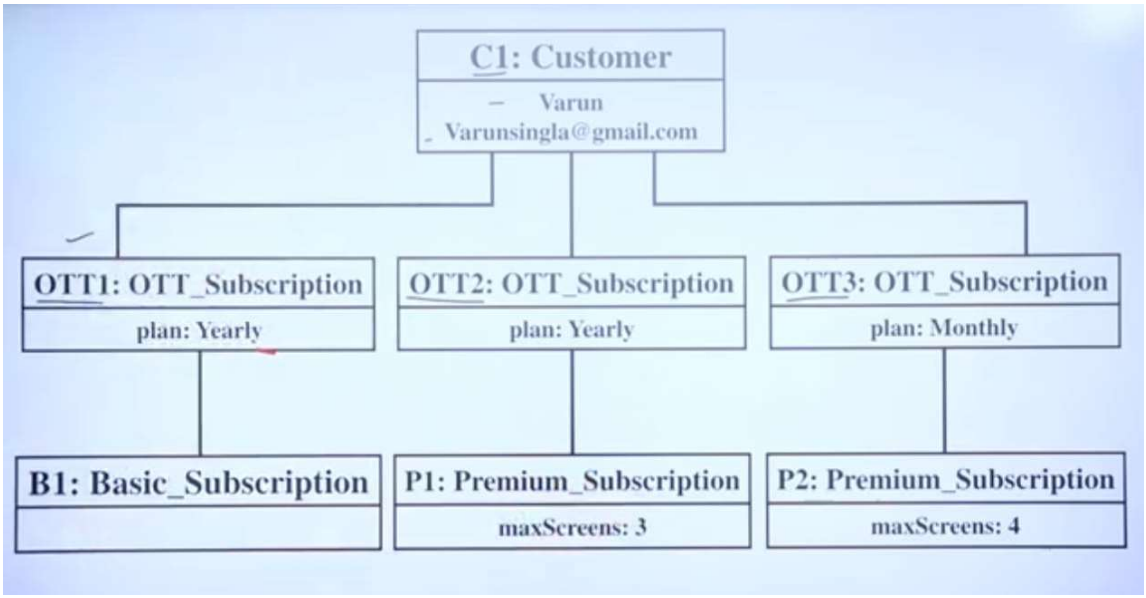


Object Reprsentation

First you need to created Class representation and class diagram then apply object representation

Object diagrams can be imagined as the snapshot of a running system at a particular moment. Let us consider an example of a running train

Object diagram shows a static view of the system at given particular time instance

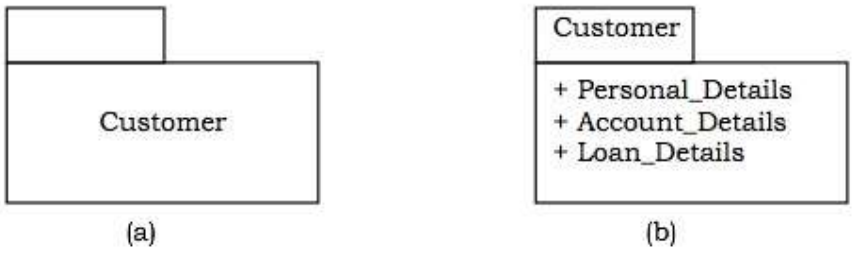


Physical Package

Package arrange number of classes, interfaces, and sub classes of the same type into group

A package is an organized group of elements. A package may contain structural things like classes, components, and other packages in it.

Notation – **Graphically, a package is represented by a tabbed folder.** A package is generally drawn with only its name. However, it may have additional details about the contents of the package. See the following figures.



it is a namespaces that organises a set of related classes and interfaces
it ia good programming practice to keep thing organised by placing related classes and interfaces into package

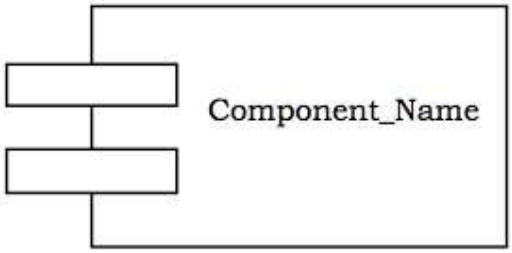
Component

A component is a physical and replaceable part of the system that conforms to and provides the realization of a set of interfaces. It represents the physical packaging of elements like classes and interfaces

Package are two Types

Pre-defined

Pre-Defined	User defined
Java.lang #bydefault Java.util #classes & interfaces Java.io Java.applet Java.awt Java.net Java.sql	package p1 package add package mypack



Document design consideration

Here are some things to consider when documenting design:

- **Document length**: Keep the document concise and not too complicated.
- **Collaboration**: Invite feedback and collaboration throughout the process.
- **Visual aids**: Use charts, diagrams, and screenshots to help guide the user.
- **Plain language**: Use plain language and short sentences and paragraphs.
- **Bullet points**: Use bullet points and number lists to make the document easier to scan.
- **Design principles**: Consider the principles of alignment, contrast, proximity, and repetition.

Design Documentation in Software Engineering

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language. The software design process can be divided into the following three levels of phases design:

Interface Design
Architectural Design
Detailed Design

S.No	Software Design Document	Module, Subpart
01.	Reference Documents	1. Existing software documentation 2. System Documentation 3. Vendor(hardware or software) documents 4. Technical reference
02.	Modules for each module	1. Processing narrative 2. Interface description 3. Design language(or other) description 4. Modules used
03.	Scope	1. System objective 2. Hardware, software and human interfaces 3. Major software functions 4. Externally defined database 5. Major design constraints, limitations
04.	Design Description	1. Data description 2. Derived program structure 3. Interface within structure
05.	Test Provisions	1. Test guidelines 2. Integration strategy 3. Special considerations
06.	Packaging	1. Special program overlay provisions 2. Transfer consideration
07.	File Structure and global data	1. External Files structure 2. Global data 3. File and data cross – reference
08.	Requirement cross-reference	1. cross-reference

Importance of Design Documentation:

1. Requirements are well understood: With proper documentation, we can remove inconsistencies and conflicts about the requirements. Requirements are well understood by every team member.

2. Architecture/Design of product: Architecture/Design documents give us a complete overview of how the product look like and better insight to the customer/user about their product.

3. New Person can also work on the project: New person to the project can very easily understand the project through documentations and start working on it. So, developers need to maintain the documentation and keep upgrading it according to the changes made in the product/software.
4. Everything is well Stated: This documentation is helpful to understand each and every working of the product. It explains each and every feature of the product/software.
5. Proper Communication: Through documentation, we have good communication with every member who is part of the project/software. Helpful in understanding role and contribution of each and every member.

Structured Analysis and Structured Design (SA/SD)

Structured Analysis	Structured Design
focuses on understanding and documenting system requirements	which transforms these requirements into a well-organized software architecture

Structured Analysis is the SA/SD methodology's initial phase, focusing on comprehending a system's requirements and constraints. The primary objective is to create a clear and precise representation of the system's functionalities and its interactions with the external environment. **Key components of Structured Analysis include:**

- **Data Flow Diagrams (DFD):** DFDs represent the flow of data within the system and how it is processed. They help visualize the data transformations and identify the input and output sources.
- **Data Dictionary:** This component provides a comprehensive catalog of all data elements used in the system, their definitions, and relationships. It ensures consistency in data representation throughout the system.
- **Entity-Relationship Diagrams (ERD):** ERDs depict the relationships between different entities within the system. They help in understanding the data structure and its dependencies.
- **Process Specifications:** These describe the processes or functions within the system, detailing their inputs, processes, and outputs. It aids in understanding the logic behind each operation.

Structured Design

Once the requirements have been thoroughly analyzed and documented, the focus shifts to Structured Design, where the goal is to transform the requirements into a structured and modular software architecture. Key components of Structured Design include:

- **Structure Charts:** Structure charts represent the modular structure of the software system, illustrating the hierarchy of modules and their interactions. **Each module** is designed to perform a specific function, promoting modularity and reusability.
- **Pseudocode:** Pseudocode is a high-level **description of the program logic**, using a combination of natural language and programming-like syntax. It helps in communicating the design logic to both technical and non-technical stakeholders.

- **Decision Tables: Decision tables are used to represent complex decision-making** processes within the system. They provide a systematic way to handle various combinations of inputs and conditions.
- **Hierarchy Charts:** Hierarchy charts illustrate the **hierarchical structure of modules, showcasing their relationships and dependencies**. This aids in understanding the organization of the software components.

Structured Design promotes a systematic and organized approach to building software systems, emphasizing clarity, maintainability, and ease of understanding.

Benefits of SA/SD

Some of the Benefits of SA/SD are discussed below:

Structured Analysis	Structured Design
focuses on understanding and documenting system requirements	which transforms these requirements into a well-organized software architecture

- **Clarity and Understanding:** SA/SD provides a clear and systematic way to understand and document system requirements, promoting a shared understanding among stakeholders.
- **Modularity and Reusability:** The modular approach of Structured Design encourages the creation of independent and reusable modules, simplifying maintenance and future enhancements.
- **Efficiency and Maintainability:** The structured nature of the methodology ensures that software systems are designed with efficiency and maintainability in mind, reducing the complexity of the overall system.
- **Communication:** SA/SD methodologies provide a common language for communication between developers, analysts, and other stakeholders, facilitating collaboration and reducing the chances of misunderstandings.

Jackson Structured Development

Jackson System Development (JSD) is a linear method of system development created by Michael A. Jackson and John Cameron in the 1980s. **It comprises the whole software life cycle directly** or providing a framework for more specialized techniques.

The main principles of JSD's work are:

- JSD method lets describe and model the real world, not specifies and not structures the function performed by the system.
- The time-ordered world must be time-ordered itself, JSD depicts the progress in the real world that models it.
- The implementation of the system is based on transformation of specification into efficient set of processes.

Jackson System Development includes three main stages, each of which is divided into steps and sub-steps:

Jackson Structured Development

Jackson System Development (JSD) is a linear method of system development created by Michael A. Jackson and John Cameron in the 1980s.

- **Modelling stage (analysis)** - includes the entity/action step and entity structures step; on this stage is created a set of **Entity Structure Diagrams (ESDs)** and are identified the entities in the system, the actions, and the attributes of the actions and entities.
- **Network stage (design)** - contains the initial model step, function step, and system timing step; **on this stage is developed a System Specification Diagram (SSD)** or a Network Diagram which use rectangles to depict the processes and diamonds to represent their relationships. This stage defines the simulation of a real world, adds the executable operations and processes, and provides the synchronization among processes and introduces the constraints.
- **Implementation stage (realization)** consists of the implementation step which converts an abstract network model into a physical system and represents it as a System Implementation Diagram (SID).

Jackson System Development (JSD) Steps

Initially, there were six steps when it was originally presented by Jackson, they were as below:

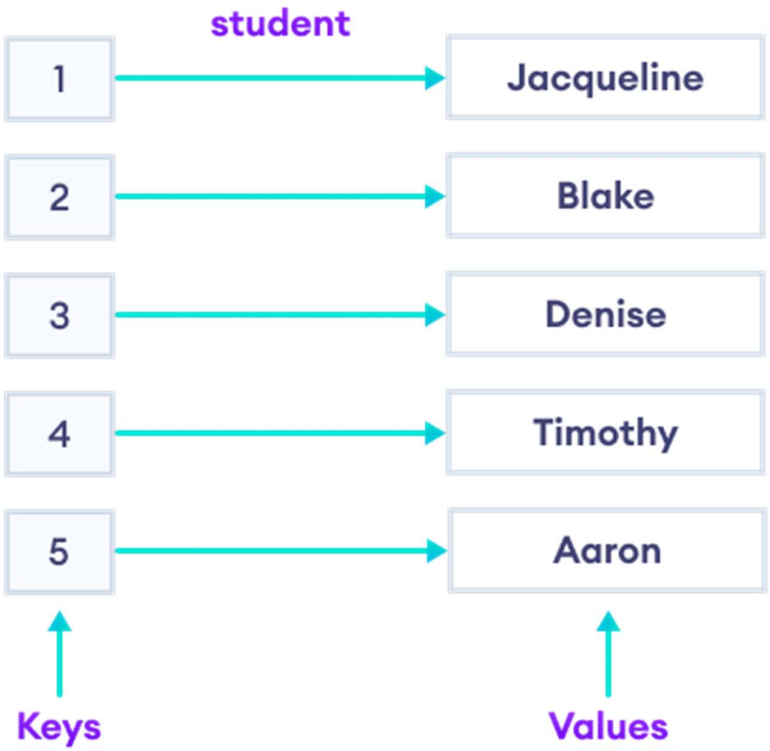
1. **Entity/action step**
2. **Initial model step**
3. **Interactive function step**
4. **Information function step**
5. **System timing step**
6. **System implementation step**

Benefits of JSD

1. It is designed to solve the real-time problems.
2. JSD modelling focuses on time.
3. It considers simultaneous processing and timing.
4. Provides functionality in the real world.
5. It is a better approach for microcode applications.

Mapping Object Oriented

Object-oriented programming (OOP) is a programming paradigm that focuses on modeling software systems as collections of interacting objects, each with its own data and behavior. Languages that support OOP provide mechanisms to define objects, classes (blueprints for objects), and the interactions between them. On the other hand, non-object-oriented languages do not adhere strictly to the OOP paradigm and may use other programming paradigms such as procedural programming or functional programming. Here's a comparison between object-oriented and non-object-oriented languages:



In order to use maps in C++, we must include the map header file in our program:
#include <map>

Create a Map

We can declare a map using the following syntax:
std::map<key_type, value_type> map_name = {{key1, value1},{key2, value2}, ...};

// create a map with integer keys and string values
std::map<int, string> student = {{1,"Jacqueline"}, {2,"Blake"}, {3,"Denise"}};

Key Characteristics:

Objects and Classes:

Objects are instances of classes that encapsulate data (attributes) and behavior (methods or functions).
Classes define the blueprint or template from which objects are created, specifying the attributes and methods that objects of that class will have.

Encapsulation:
Encapsulation involves bundling data (attributes) and methods (functions or procedures) that operate on the data into a single unit (class).
Objects interact with each other through well-defined interfaces, hiding internal implementation details.

Inheritance:
Inheritance allows classes to inherit attributes and behaviors from parent classes (superclasses).
It promotes code reuse and enables hierarchical relationships between classes.

Polymorphism:
Polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling flexibility and extensibility in code design.
It supports method overriding (redefining a method in a subclass) and method overloading (defining multiple methods with the same name but different parameters).

Non-Object-Oriented Languages
Key Characteristics:

Procedural Programming:
Procedural languages focus on procedures (functions or subroutines) that operate on data.
Data and procedures are separate, and the emphasis is on step-by-step procedures for solving problems.
Functional Programming:
Functional programming is a programming paradigm that treats computation as the **evaluation of mathematical functions** and avoids changing state and mutable data.
Examples of Non-Object-Oriented Languages:

C: **A procedural language known for its efficiency, low-level memory access, and widespread use in system programming and embedded systems.**
Fortran: Originally designed for scientific and engineering calculations, it supports procedural programming with strong mathematical capabilities.
Lisp: A functional programming language with a focus on symbolic computation, list processing, and macros.

Comparison

- **Abstraction**: OOP provides a higher level of abstraction by modeling **real-world entities as objects**, while non-OOP languages may focus more on procedures or functions.
- **Code Reusability**: OOP promotes code reusability through inheritance and polymorphism, whereas non-OOP languages achieve reuse through modular programming and functional composition.
- **Complexity**: OOP can introduce additional complexity due to its concepts like inheritance hierarchies and polymorphic behavior. Non-OOP languages may offer simpler approaches to problem-solving, depending on the paradigm used.
- **Community and Ecosystem**: OOP languages like Java and Python have large communities and extensive libraries/frameworks. Non-OOP languages may have specialized communities based on their specific paradigms and applications.

Conclusion

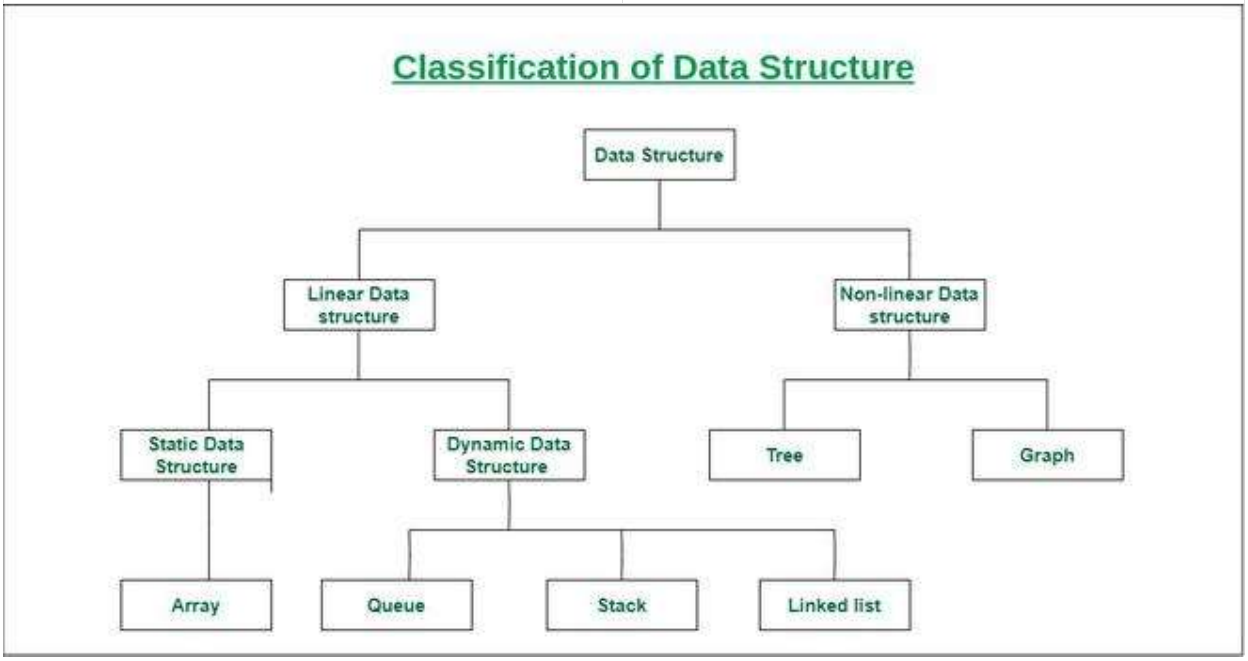
The choice between using an object-oriented or non-object-oriented language depends **on factors such as the nature of the problem domain, project requirements, performance considerations, and developer preference**. Object-oriented languages provide powerful tools for modeling complex systems and promoting code reuse, while non-object-oriented languages offer alternative paradigms suited to specific types of applications and programming styles.

Translating classes into data structures

A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need. Most importantly, data structures frame the organization of information so that machines and humans can better understand it.

Classification of Data Structure
Linear Data Structure
1) Static data structure
2) Dynamic data structure
Non-linear Data Structure

Data Type	Data Structure
The data type is the form of a variable to which a value can be assigned. It defines that the particular variable will assign the values of the given data type only.	Data structure is a collection of different kinds of data. That entire data can be represented using an object and can be used throughout the program.
It can hold value but not data. Therefore, it is dataless.	It can hold multiple types of data within a single object.
The implementation of a data type is known as abstract implementation.	Data structure implementation is known as concrete implementation.



- **Linear data structure:** Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
  Examples of linear data structures are array, stack, queue, linked list, etc.
    - **Static data structure: Static data structure has a fixed memory size.** It is easier to access the elements in a static data structure.
      An example of this data structure is an array.

- **Dynamic data structure:** In the dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
  Examples of this data structure are queue, stack, etc.
-

- **Non-linear data structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.
  Examples of non-linear data structures are trees and graphs.

Need Of Data structure :
The structure of the data and the synthesis of the algorithm are relative to each other. Data presentation must be easy to understand so the developer, as well as the user, can make an efficient implementation of the operation.
Data structures provide an easy way of organizing, retrieving, managing, and storing data.
Here is a list of the needs for data.

Data structure modification is easy.
It requires less time.
Save storage memory space.
Data representation is easy.
Easy access to the large database.

Class

In object-oriented programming, a class is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods). The user-defined objects are created using the class keyword.

n object-oriented programming (OOP), a class is a user-defined data type that acts as a blueprint for creating objects:

What is a Class in C++?

A class is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.
For Example: Consider the Class of **Cars**. There may be many cars with different names and brands but all of them will share some common properties

like all of them will have *4 wheels*, *Speed Limit*, *Mileage range,* etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties.

- **A Class is a user-defined data type that has data members and member functions.**
- **Data members are the data variables and member functions are the functions used to manipulate these variables together, these data members and member functions define the properties and behaviour of the objects in a Class.**

But we cannot use the class as it is. We first have to create an object of the class to use its features. An **Object** is an instance of a Class.

- **Definition**

  A class is a template that defines the variables and methods for a specific type of object.

- **Structure**

  A class has data members and member functions. Data members are the variables, and member functions are the functions that manipulate those variables.

- **Objects**

  An object is a specific instance of a class, and contains real values instead of variables.

- **Subclasses**

  A class can have subclasses that inherit some or all of the class's characteristics. The class is the superclass in relation to its subclasses.

- **Class hierarchy**

  The structure of a class and its subclasses is called the class hierarchy.

- **Benefits**

  Classes encourage modularity, reusability, and code organization, which makes it easier to maintain complex systems.

Abstraction
Data Abstraction is a providing only the essential details to the outside world and hiding the internal details.

What is abstract class
in abstract class it does now allow object and when virtual function is declared then it is abstract class

Constructor
constructor can be declared in two ways.
within class
outside class

```
#include <iostream>
using namespace std;
class student
{
	int rno;
	string name;
	public: student()
			{
				cout<<"Enter the RollNo:";
				cin>>rno;
				cout<<"Enter the Name:";
				cin>>name;
			}

	void display()
	{
		cout <<"Name "<<name<<" "<<"Roll No "<<rno;
	}
};

main()
{
	student s;
	s.display();
}
```

Outside the class:

```
#include <iostream>
using namespace std;
class student
{
	int rno;
	string name;
public:
	student();

	void display();
```

```
};

student::student()
        {
        cout<<"Enter the RollNo: ";
        cin>>rno;
        cout<<"Enter the Name: ";
        cin>>name;
        }

void student::display()
{
        cout<<"\nName "<<name;
        cout<<"\nrollNo. "<<rno;
}
int main()
{
        student s;
        s.display();
        return 0;

}
```

Abstraction

Abstraction refers to providing only essential information about the data to the outside world, ignoring unnecessary details or implementation.

Types of Abstraction:

1. **Data abstraction –** This type only shows the required information about the data and ignores unnecessary details.
2. **Control Abstraction –** This type only shows the required information about the implementation and ignores unnecessary details.

```
#include <iostream>
using namespace std;

class implementAbstraction {
private:
   int a, b;

public:
   // method to set values of
   // private members
```

```
    void set(int x, int y)
    {
        a = x;
        b = y;
    }

    void display()
    {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};

int main()
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}
```

Abstract Class

```
#include <iostream>
using namespace std;
class student
{
        public:
        virtual void f1()=0; // pure virtal function

        void f2() {

                cout<< "I am F2() Function "<< endl;

        }

};
class abc:public student {
        public:
void f1(){
        cout<<"I am f1 in abc class"<<endl;
}
```

```
};

int main()
{
	//student s;
	abc obj;
	obj.f1();
	obj.f2();
}
```

Example 2 :

```
#include <iostream>
using namespace std;
class Bank {
	public:
		float a, interest, totBalance;
		int acno;
		virtual void calInterest(float balance)=0;

		void showAcno(int m)
		{
			acno=m;
			cout<<"Account Number "<<acno<<endl;

		}

};

class sbi:public Bank {
	public:
		void calInterest(float balance)
		{
			a=balance;
			interest=a*5/100;
			totBalance=interest+balance;
			cout<<"Interest Amount "<<interest<<endl;
			cout<<"Total amount "<<totBalance <<endl;
		}
};
```

```
int main()
{
	sbi obj;
	obj.showAcno(1254);
	obj.calInterest(1000);
	return 0;
}
```

- **Namespace :**Using namespace, you can define the space or context in which identifiers are defined i.e. variable, method, classes. In essence, a namespace defines a scope.

Advantage of Namespace to avoid name collision.

- Example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

```
};
//when do not require the function definition then it'll be pure virtual function
//this is necesary to use function overruding to avoid abstract class in derived class
//when do not require the function definition then it'll be pure virtual function
int main()
{
	sbi s;
	s.showAcno(2512);
	s.calInterest(10000);
	return 0;

}
```

Code Reusability
Inheritance for "Which feature of OOPS illustrated the code reusability"

Inheritance allows you to create a new class (subclass or derived class) that inherits properties and behaviors from an existing class (superclass or base class). This promotes code reusability by allowing you to reuse the code from the existing class in the new class without duplicating it.

Encapsulation
Binding the data into single group

Protect The data
Improves the security
Simplifies data hiding