Understanding system:

Our environment majorly surrounded by systems isn't it!?. We can divide each system as sub-systems, with respective their behaviors.Let us takes the body system of Human. Nerves System, Digesting system, Muscular system etc. They classified with their behavioral thing. each sub-system takes the job of doing a certain thing. And if we get into deep, those systems are even segmented as tissues and cells.It is just a simple analogy, our universe, and nature, all things are abstracted by the above aspect. So programming languages and system designing also
represented by this perspective. So let me assume that you have a basic conceptual idea about the system. So let us list the properties of the system.

1.Entity- A subsystem
2. Object- entity lives in sub-system
3.Hierarchy- Chaining of objects or entities
4.Abstraction- Hiding the internal details of the entity by the top layers of sub-systems.

Let us discuss the layers in details

Understanding Objects:

Objects are unit in object-oriented programming, like a cell in a human body or atoms in a molecule structure.these cells are formed together by individual groups and doing their respective jobs to alive a body. Like this, the group of objects formed together by their state and behavior and working along the code. Most of the textbooks say that every object has a state and behavior.
Let us examine these characteristics via a simple aspect. For instance, consider a class which can hold the details of the student, it can store name, roll no, marks, and attendance. You can create several numbers of objects for that class.
First, two attributes(name, roll no) are essentials, that every student must have it you can't change.
So you determine the constructor to set those values. Of course, once it is initialized, you can't change it. That is the state of the object.
the state can't be changed, such an allocated memory for an object.

Constructor-Determine the state
Destructor-destroys the state. (Typically we haven't in java-Because JVM contains garbage collection mechanism)

public class Student {

```
 String name;
 int rollno,marks,attendance;

    public Student(String name, int rollno) {
        this.name = name;
        this.rollno = rollno;
    }
    public String getName() {
        return name;
    }
    public int getMarks() {
        return marks;
    }

    public void setMarks(int marks) {
        this.marks = marks;
    }

    public int getAttendance() {
        return attendance;
    }

    public void setAttendance(int attendance) {
        this.attendance = attendance;
    }


}
```

Abstraction:

Object orientation's one of the most successive ideas is an abstraction. Technically means that "bringing simplicity from complexity".
Our world uses abstraction to manage chaos. For example, take an analogy of real-time software. Let us takes simulation software, or similar to that software.
This is a software where you can simulate the virtual electronic components and other virtual devices. Consider an interface called IntegratedChip. Of course, all are IC components, like gates, and processors. Here are the methods of IntegratedChip

1.setPins(int count);
2.drawIC();
3.giveVoltage(double voltage)

4.takeOutput();

AND gate:

setPins - 3
drawIC - draw on canvas
giveVoltage - 5v;
output-input1&input2

D flipFlop:

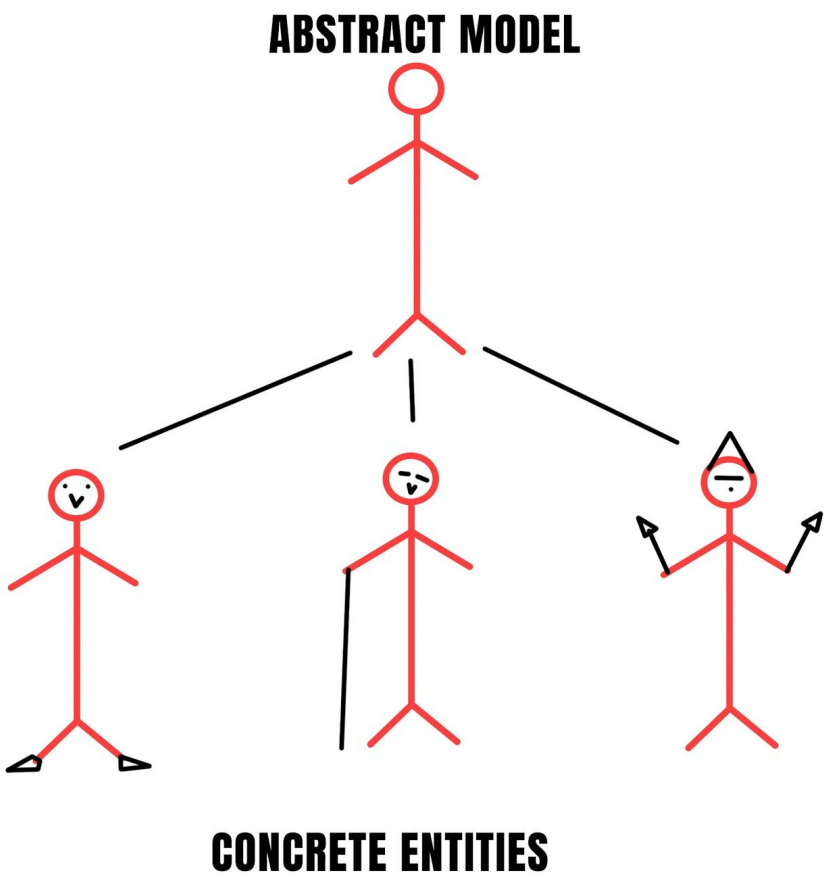
setPins - 5
drawIC- draw on canvas
giveVoltage- 10v;
output-other logics;

Any ic's which implements the IntegratedChip interface, must implement and give an initialization for these methods. In other words, IntegratedChip is an abstract idea of all chips(which contains only the declaration-idea, not contains how to do that). Of course, everyone has a different characteristic such as input voltage, and pin number.

Some chips have less than other ones. But the main idea behind the creation of IntegratedChip is same for all. And that is the abstraction.Interfaces and abstract classes are a fundamental tool in Java for implementing abstraction. The main thing is there is no object for abstract classes.

Though abstraction, represents the above meanings, the term varies with the perspective of the objective. For instance, we can create the abstraction for objects,
by their action, model, are by their behavioral controls. If a man, is a generic abstraction, then male and females are concrete entities, and by other perspectives, good and evil are concreted. So the abstraction varies with the perspective, one should carefully mind it while design

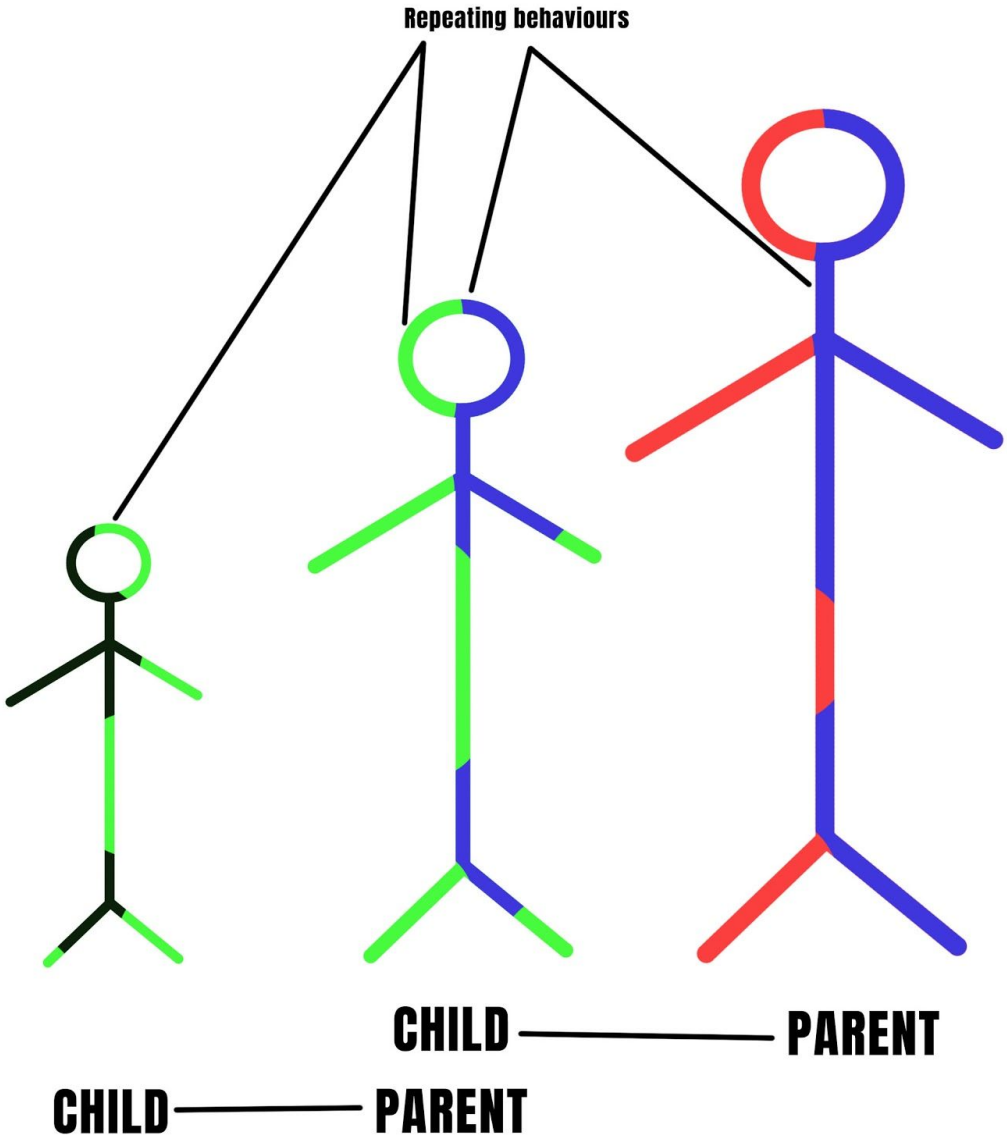
object-oriented software or hardware.



Inheritance:

In the real world, one object is a product of another one. such a father and son, son has a behavior of father, and father has grandfathers.
Like a subclass and superclass, a class can inherit another class to get its behavior. If you ever worked in Java,
you may possibly know the threads, the thread is runnable, waitable, delayable, and noticeable. If you want your class with these characteristics,
what do you do?. you need to get the behavior of the class. so you inherit the class using "extends" keyword, like - 'your class extends Thread.'
There is Garganta amount of example for this. Of course, we always relate one object with another one. In a swing GUI development, if we extend the JFrame,

we can treat that class like a frame. So shortly, the subclass inherits the behavior from the superclass. And another example, in a social media group, considers the principle of members. Group admin also a member, so group member inherits the attributes of a member also. Of course, admin also has its special attributes.



Encapsulation:

The class contains methods, like a shell. But encapsulation concept muchly focuses on the access protection to them. Many of the programmers, misunderstood this term "access protection", as a security thing, like passwords and codes can be protected here. No, they are networking things, encapsulation is different. Of, course they are used for secure the access of methods, inside the class, or direct classes, and variables. Here is the question, why?. Ok, let us take an example. Here is the class for drawing the right angle triangle. in the class, there are three methods, draw(), setAngle(),drawLines(). In that class, only draw() is accessible from outside, because the user may misuse the variables like the angle, or the invoking of other methods, in an inappropriate way, this may collapse the purpose of that class. Imagine if the user knowingly, or unknowingly changes the angle to 60 deg, will it draw right angle triangle anymore?. In shortly, a user only allowed to get the interface access for those classes, they don't need to
care about their internals and working.

```
class RightAngleTriangle{
private int angle=90;
 draw(){
     canvasDraw(angle);
 }
}

class main{
  rightangleInstance.draw();
}
```

Here angle is private because if the user changes it, it will damage the whole purpose for that class or instance.

So encapsulation is a solid method for differentiating what user needs, and what he doesn't need. It is like a shield around a powerful engine. It products
internal parts as well as outer elements. The purpose is getting the benefit from it.

Polymorphism:

Consider a rectangle, you can draw it from two ways, one by length and breath another one from the crossectional length. can you create two methods for that, yes but what if the user can use the same method name?
Of course, it is method overloading. drawRec(int l,int h) and drawRec(int c). Is polymorphism only about this? absolutely not. It can provide a powerful run time polymorphism by the usage of interfaces.
Consider the below example,

```
interface Shape{
  void draw();
}
class square{
draw(){
//draw square
}
}

class circle{
draw(){
//draw circle
}
}
```

if the objects of shape listed together and iterate to invoke draw(), method. The JVM automatically identify the object and invoke their individual method correctly.
This is a powerful polymorphic tool in terms of object orientation. let us take the same example from the above topic abstraction, all it's are Shape, so.

```
Shape sq=new square();
Shape ci=new Circle();
```

If the canvas class contains the list of Shapes selected by the user, then we need to draw it. Through the iteration, there is a simple skeleton which is enough to draw the whole logic, an example is given below

```
for(int i=0;i<selectedShapeList.size();i++)
{
selectedShapeList[i].draw();
}
```

here there is no need to call separate draw methods for different shapes like square, circle or rec. We can only call its generic method(it shouldn't be null).

Design patterns:

Object-oriented design patterns mostly describe the relationship between objects and classes, it involves the philosophy of creating and manipulating objects.
I am going to skip the content history of design patterns and GoF(Gang of four-Authours of the first original book about design patterns). These patterns really do a favor in complex software development, moreover it gives meaning to the object-oriented design.

Factory design pattern

This factory gives the abstract platform to create the objects for the client classes, It has access to create the objects to the other product classes, and based on requirements

It supplies the object. In simple terms, factory class provides the object for product classes. It works like a factory, which creates many types of products. Let us see the example here

```
package factory;

import factory.Man.ManCategory;
import static factory.Man.ManCategory.MEDIUM;
import static factory.Man.ManCategory.SHORT;
import static factory.Man.ManCategory.TALL;

/**
 *
 * @author AMUTHAN
 */
 abstract class Man {
    ManCategory type;

    Man(ManCategory type){
    this.type=type;
    }

  enum ManCategory{
   TALL,MEDIUM,SHORT
  }

  public abstract void createMan();

}
 class indian extends Man{

    public indian() {
       super(ManCategory.MEDIUM);
```

```
    }

    @Override
    public void createMan() {
        System.out.println("creating indian");
     }

}
class african extends Man{

    public african() {
       super(Man.ManCategory.TALL);
    }

    @Override
    public void createMan() {
        System.out.println("creating african");
     }

}
class chinease extends Man{

    public chinease() {
       super(Man.ManCategory.SHORT);
    }

    @Override
    public void createMan() {
        System.out.println("creating chinease");
     }

}

class ManFactory{
 //This factory will create and return the object for us, based on our requirement
    //i.e if-> return that or return someother
    //here i use enumerations, you can use strings, integers also
    //i.e if(i==5){retrun that} or if(string.equals("")){return that}
  public Man createMen(ManCategory type){

  if(type==TALL){
   return new african();
  }
```

```
  if(type==MEDIUM){
   return new indian();
  }
  if(type==SHORT){
   return new chinease();
  }
  else{return null;}
  }

}
public class FactoryDemo{
public static void main(String[] args){
   ManFactory fac=new ManFactory();

   fac.createMen(TALL).createMan();

   fac.createMen(MEDIUM).createMan();

   fac.createMen(SHORT).createMan();


}

}
```

when there is so much of subclasses, or product classes to consider in your main class, this factory method really helps you to give one layer of abstraction. In simple terms,
this method can be also an example of inheritance and runtime polymorphism.In the above code, a creator is a man, which is generally of the African,Indian, Chinese. They are all concrete implementation of the man.

Singleton design pattern

When you want to allow the class to provide only one instance or object, you better implement the singleton design pattern. When the single instance must be extensible by child-classing, and clients should be able to use an extended instance without changing their code. For example, imagine you have a website that accesses some database. If the client can get
multiple instances for the class, it may lead to concurrency problems and data corruption. In this time singleton design pattern is really useful.

```
public class Singleton {
    //creating object, only one time, in other words only one instance for the class
    private static Singleton obj;
```

```
    public static synchronized Singleton getInstance(){

    if(obj==null){
      obj=new Singleton();
    }

    return obj;
    }
}

public class SingletonDemo {
    public static void main(String[] args) {
        Singleton s=Singleton.getInstance();
        Singleton s1=Singleton.getInstance();

        //it prints CREATED
        if(s!=null){
        System.out.println("object s is created...!");
        }

        else{System.out.println("object s not created...!because already created");}

        //it prints NOT CREATED
        if(s1!=null){
        System.out.println("object s1 is created...!");
        }
        else{System.out.println("object s1 not created...!because already created");}
    }
}
```

As per the above code, you can find that the singleton class can't allow more than one instance.

Facade design pattern

Another important design pattern is the facade, which gives a top layer of abstraction for the subsystem. Let us define the subsystem here.
when we want to write bytes to the file, we usually do open the file, get the output stream for that file, and create the output stream writer, finally, we write on it. After writing, we need to close one by one. Same goes for every operation. The problem is, user needs to manage complexity through the entire
process. what if we create two concrete methods for start and stop?. Here the facade design pattern comes. a facade design pattern provides the interface for
reducing the process steps.

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


        //facade pattern's approach
        Facade facade=new Facade();
        facade.createParts();
        facade.makeFunctions();

        System.out.println();

        //traditional approach without pattern
         Man m=new Man();
         m.createHand();
         m.createHead();
         m.createLeg();
         Functions f=new Functions();
         f.createEatingFunc();
         f.createWalkingFunc();
         f.createSpeakingFunc();
         f.getWalkingFunc().walk(m.getLeg());
         f.getSpeakingFunc().speak(m.getHead());
         f.getEatingFunc().eat(m.getHand());
    }
}

public class Facade {
   //this pattern create simplicity while handling multiple subsytems;
    Man m;
    Functions f;
  public void createParts(){
    m=new Man();
    m.createHand();
    m.createHead();
    m.createLeg();
  }
  public void makeFunctions(){
   f=new Functions();
   f.createEatingFunc();
   f.createSpeakingFunc();
   f.createWalkingFunc();
   f.getWalkingFunc().walk(m.getLeg());
```

```
    f.getSpeakingFunc().speak(m.getHead());
    f.getEatingFunc().eat(m.getHand());
   }
}
```

It may give a small problem to the user that, the process can't customized. The process will be started or stopped, with knowledge and methods of the facade class.

Composite pattern

This pattern can be treated as, hierarchical pattern. It is one of the behavioral patterns, which compose the behavior of an object. The composite means “putting together” and this is what this design pattern is all about. One object composed to another one and that object encapsulate by another object. Moreover, we use it many times, without the knowledge of the pattern
name. And it is one of the ways to implement a tree data structure(node and leaf).In Composite Pattern, elements with children are called as Nodes, and elements without children are called as Leafs.

```
//this is composite interface
interface BodyPart{
 public void showDetails();
}
class fingers implements BodyPart{
    int fingerNo;
    fingers(int a){
    fingerNo=a;
    }
    int getFingNo(){
    return this.fingerNo;
    }
    @Override
    public void showDetails() {
        System.out.println("finger "+fingerNo);
              }
}
class Hand implements BodyPart{
    List<fingers> fing=new ArrayList<>(5);
    int handno;
    public Hand(int no){
        handno=no;
    }
```

```
    public void addfing(fingers f){
        fing.add(f);
    }
    public void showDetails() {
        System.out.println("hand no "+handno);
       for(int i=0;i<fing.size();i++) {
        System.out.println("finger no "+fing.get(i).fingerNo);
       }
     }


}
class Leg implements BodyPart{
    List<fingers> fing=new ArrayList<>(5);
    int legno;
    public Leg(int no){
        legno=no;
    }
    public void addfing(fingers f){
        fing.add(f);
    }
    public void showDetails() {
        System.out.println("hand no "+legno);
       for(int i=0;i<fing.size();i++) {
        System.out.println("finger no "+fing.get(i).fingerNo);
       }
     }


}
class Body implements BodyPart{
    List<Hand> hands=new ArrayList<>(2);
    List<Leg> legs=new ArrayList<>(2);
    public void add(Hand h){
    hands.add(h);
    }
    public void add(Leg l){
    legs.add(l);
    }
    @Override
    public void showDetails() {

       for(int i=0;i<hands.size();i++) {
```

```
            System.out.println("hand no "+hands.get(i).handno);
        for(int j=0;j<hands.get(i).fing.size();j++) {
            System.out.println("finger no "+hands.get(i).fing.get(j).getFingNo());
           }
       }

       for(int i=0;i<legs.size();i++) {
            System.out.println("leg no "+legs.get(i).legno);
        for(int j=0;j<legs.get(i).fing.size();j++) {
            System.out.println("finger no "+legs.get(i).fing.get(j).getFingNo());
           }
       }
    }


}




public class CompositeDemo {

    public static void main(String[] args){
        //for hand 1
      fingers f1=new fingers(1);
      fingers f2=new fingers(2);
      fingers f3=new fingers(3);
      fingers f4=new fingers(4);
      fingers f5=new fingers(5);
       Hand leftHand=new Hand(1);
       leftHand.addfing(f1);leftHand.addfing(f2);leftHand.addfing(f3);leftHand.addfing(f4);
       leftHand.addfing(f5);
       //for hand 2
       fingers f11=new fingers(6);
      fingers f12=new fingers(7);
      fingers f13=new fingers(8);
      fingers f14=new fingers(9);
      fingers f15=new fingers(10);
       Hand rightHand=new Hand(2);
       rightHand.addfing(f11);rightHand.addfing(f12);rightHand.addfing(f13);
       rightHand.addfing(f14);
       rightHand.addfing(f15);
      //for leg1
```

```
      fingers f21=new fingers(11);
     fingers f22=new fingers(12);
     fingers f23=new fingers(13);
     fingers f24=new fingers(14);
     fingers f25=new fingers(15);
     Leg leftLeg=new Leg(1);
     leftLeg.addfing(f21);leftLeg.addfing(f22);leftLeg.addfing(f23);
     leftLeg.addfing(f24);leftLeg.addfing(f25);
     //for leg2
     fingers f31=new fingers(16);
     fingers f32=new fingers(17);
     fingers f33=new fingers(18);
     fingers f34=new fingers(19);
     fingers f35=new fingers(20);
     Leg rightLeg=new Leg(2);
     rightLeg.addfing(f31);rightLeg.addfing(f32);rightLeg.addfing(f33);
     rightLeg.addfing(f34);rightLeg.addfing(f35);

     Body b=new Body();
     b.add(leftHand);
     b.add(rightHand);
     b.add(leftLeg);
     b.add(rightLeg);

     b.showDetails();

   }

}
```

this pattern is about creating a hierarchy of the objects and composing more and more leaves in a component.
The Composite pattern can be implemented anywhere you have the hierarchical nature of the system or a subsystem and you
want to treat individual objects and compositions of objects uniformly. here the "uniform" term tells that body part interface.
so these all classes are uniformly composed.

Builder pattern

Builder pattern provides the class for building, desired products for a client. It is similar to the factory method, but if you see carefully, there are more than
one difference between them.

```
abstract class man{
 String type;
 String height;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getHeight() {
        return height;
    }

    public void setHeight(String height) {
        this.height = height;
    }
    public String toString(){
    return this.type+" "+this.height;
    }

}
interface manbuilder{
   void setType();
   void setHeight();
   man getMan();
}
class mencreater implements manbuilder{
    man m;
    mencreater(){this.m=new man() {};}
    @Override
    public void setType() {
      m.setType("male");
   }

    @Override
    public void setHeight() {
      m.setHeight("tall");
   }
```

```
    @Override
    public man getMan() {
        return this.m;
    }

}
class womencreater implements manbuilder{
    man m;
    womencreater(){this.m=new man() {};}
    @Override
    public void setType() {
      m.setType("female");
   }

    @Override
    public void setHeight() {
      m.setHeight("medium");
   }
 @Override
    public man getMan() {
        return this.m;
    }
}

class builderclass{
manbuilder b;
builderclass(manbuilder b){
this.b=b;
}
public void getType(){
System.out.println(b.getMan());
}
public void build(){
  this.b.setHeight();
  this.b.setType();
}
public man getMan(){
return this.b.getMan();
}

}

public class BuilderDemo {
```

```
    public static void main(String[] args){
        manbuilder b=new mencreater();
        builderclass b1=new builderclass(b);
        b1.build();
        man gettedMan=b1.getMan();
        //this will print male, tall
        System.out.println(gettedMan);

        manbuilder b2=new womencreater();
        builderclass b3=new builderclass(b2);
        b3.build();
        man gettedMan1=b3.getMan();
        //this will print female, medium
        System.out.println(gettedMan1);


    }

}
```

here the products are, male and female, both have an interface of manbuilder. So the main concept plays the major role here,
which is polymorphism and in factory method which is not. A client object can create an instance of a concrete builder and invoke the set of methods required to construct different parts of the final object.

Implementation in C:

Remember oop is just a generic concept, that you can implement it in any other non-object oriented languages, It will be a better practice that if we implement in c language. Before we entering the concept, let us talk about C. C is a structure oriented language, which operated by the bunch of functions. By dissecting tasks, by a small group, and encapsulate them as functions makes it very sensible way to code. For example let us takes the model traditional class template, which is implemented in c.

```
#ifndef STUDENTCLASS_H
#define STUDENTCLASS_H

typedef struct student{
    int rollno;
    int attendance;
    char* student;
```

```
}Student;

void constructStudent(Student* st,int roll,char* name,int atten){
   st->attendance=atten;
   st->rollno=roll;
   st->student=name;

}

//sample setter and getters for roll no..
void setRoll(Student* st,int roll){
   st->rollno=roll;
}

int getRoll(Student* st){
   return st->rollno;
}

//
#endif /* STUDENTCLASS_H */
```

and the main file

```
Student s1;
   char* name="amuthan";
   constructStudent(&s1,101,name,98);
   printf("student roll no is %d..",getRoll(&s1));
```

which will give the output of,

```
student roll no is 101..
```

Let me explain the above model, the attribute sets or carefully, encapsulated by the structure, and functions are defined in the header file. In this way, encapsulation can be done in the c language.

Inheritance:

In the harmony of c language, inheritance can be implemented by passing a structure into another structure. Here shape is a superclass, and rectangle is another child class. You have separate methods for them, and you can access the attributes of superclass from child also.

```
typedef struct shape{
    int height;
    int width;
}Shape;

void constructSuper(Shape* super,int sh,int sw){
    super->height=sh;
    super->width=sw;
}
void setHeights(Shape* s,int h){
    s->height=h;
}
void setWidths(Shape* s,int w){
    s->width=w;
}
int getHeights(Shape* s){
    return s->height;
}
int getWidths(Shape* s){
    return s->width;
}

#endif /* SHAPE_H */
```

in sub-resource c file(child)

```
#include "Shape.h"


typedef struct Rectangle{
    struct Shape* parent;
    int height;
    int width;
} Rectangler;

void construct(Shape* super,Rectangler* s,int h,int w,int sh,int sw ){
    s->parent=super;
    s->height=h;
    s->width=w;
    constructSuper(super,sh,sw);
```

```
}

void setHeight(Rectangler *s,int h){
   s->height=h;
}
void setWidth(Rectangler *s,int w){
   s->width=w;
}
int getHeight(Rectangler *s){
   return s->height;
}
int getWidth(Rectangler *s){
   return s->width;
}

#endif /* RECTANGLE_H */
```

in main.c

```
 //for inheritance
   Rectangler rec;
   Shape super;

   construct(&super,&rec,10,10,20,20);
   setHeight(&rec,40);
   //setHeight(rec.parent,50);

   printf("height of sub and super classes..%d,%d",getHeight(&rec),getHeights(&super));
```

the output will be..

the height of sub and superclasses..40,20

If you read carefully, we have that constructor for rectangler, which call it's parent's(Shape's) first and then it's attributed. By the above method, inheritance can be achieved approximately.

Polymorphism:

By the effective using of pointers, we can achieve run time polymorphism in c language, In the previous chapter we see the concept of polymorphism, to remember, let us see some run time

situations. If the user picked up many shapes like square, rectangle, circle etc..., if we need to list their areas, we can't call everyone's individual methods every time. And it also becomes complex because the user's selection will we vary each time. So we need a simple, common interface to all. And from there, we can achieve this, for explanation
see the code

```
typedef struct square_{
    char padd[4];
    int width;
    int (*area)(void*);
}Square;
typedef struct rec_{
    int width;
    int length;
    int (*area)(void*);
}Rectangle;
typedef struct circle_{
    char padd[4];
    int radius;
    int (*area)(void*);
}Circle;

typedef struct shape_{
    char padd[4];
    int width;
    int (*area)(void*);
}Shape;

int squ_area(void* a){
    Square *s=(Square*) a;
    return (s->width)*(s->width);
}
int rec_area(void* a){
    Rectangle *r=(Rectangle*) a;
    return (r->length)*(r->width);
}
int cir_area(void* a){
    Circle *c=(Circle*) a;
    int r=c->radius;
    int ar=(22*r*r/7);
    return ar;
}
```

```
//in the main
  Square s1;
    Rectangle r1;
    Circle c1;

    s1.area=squ_area;
    r1.area=rec_area;
    c1.area=cir_area;
    s1.width=10;
    r1.length=9;
    r1.width=7;
    c1.radius=7;

    Shape* arr[3];
    arr[0]=(Shape*)(&s1);
     arr[1]=(Shape*)(&r1);
      arr[2]=(Shape*)(&c1);

for(int i=0;i<3;i++){
void* ref=(void*)arr[i];
int area=(arr[i]->area)(arr[i]);
printf("area of shape ,%d",area);}
```

The//output will be

area of a shape, 100 area of shape, 63 area of shape, 154

Here we create three separate structures and interface them in a common structure called Shape.
We typecast all structures to the shape. Typecasting will only succeed if all have a similar memory model. for that reason, we add padding memory totally the extra memory.