OOPS

Version - 2.01
31st Mar 2005

Table Of Contents

1. Introduction to OOPs

Object-oriented programming (OOP) has become exceedingly popular in the past few years. Software producers rush to release object –oriented versions of their products (OOP is a concept not a language!)
For many people the transition from procedural programming to object-oriented programming is not easy. In fact, this transition is often accompanied by quite a bit of anxiety because C++ is surrounded by a certain aura that makes it inaccessible.

The goal of this course is to help you to gain an understanding of the fundamental concepts driving Object Oriented Programming (OOP) in a quick and painless way. They let you begin thinking in an "object oriented way". Once you understand the fundamentals, learning any new object oriented language is relatively straightforward because you will have a framework on which to attach other details, as you need them. OOP used correctly, it can dramatically improve your productivity as a programmer

2. Benefits of OOPs

Why Object Oriented Languages?

People who are new to any object oriented language or who are trying to learn about it from books, often have two major questions:

- "Everything I read always has this crazy vocabulary–'encapsulation', 'inheritance', 'virtual functions', 'classes', 'overloading', 'friends'-- Where is all of this stuff coming from?" and
- "This language–and object oriented programming in general-obviously involve a major mental shift, so how do I learn to think in a C++ way?"

Both of these questions can be answered, and the design of OOP as a whole is much easier to swallow, if you know what the designers of C++(the first full-fledged object oriented language) were trying to accomplish when they created the language. If you understand why the designers made the choices they did, and why they designed certain features into the language, then it is much easier to understand the language itself.

Lets first see problems with Structured Programming:-
As programs grow larger and compiles, the structured programming approach fails. Analyzing the reasons for these failures reveals that there are weaknesses in the procedural paradigm itself. No matter how well the structured programming approach is implemented, large programs become excessively complex.

In a procedural language, the emphasis is on doing things – read the keyboard, check for errors and so on. i.e. emphasis is on action. Data is given a second class treatment.

In procedural languages, we declare global variables; these variables can be accessed by all functions. These functions can perform various operations on the data. You may declare local variables, but they cannot be accessed by all functions.

Now suppose a new programmer is hired to write a function to analyze the data in a certain way. Unfamiliar with the program, the programmer may create a function that accidentally corrupts the data. This is possible because the function has complete access to the data.

Another problem is that since many functions access the same data, the way the data is stored becomes critical. The arrangement of the data cannot be changed without modifying all the functions that access it.

What is needed is a way to restrict access to the data, to hide it from all but a few critical functions. This will protect the data and simplify maintenance.

Procedural programs are often difficult to design. The problem is that their chief components - functions and data structures – do not model the real world very well.

With traditional languages, there is a problem of creating new data types. You cannot bundle both X and Y into a single variable called Point and then add and subtract values.

Unlike structures approach the OOP approach always model the real world well. OOP always talk in terms of the objects, wherein the procedure oriented approach talk in terms of instructions or steps to be carried out. In simple word, OOP give more emphasis on data and procedure oriented approach give more emphasis to instructions. To take a simple example, suppose I want to tell somebody , how did I get a book from library.

The procedure oriented approach would expect to write down following steps:-

1) Start from your desk.
2) Start walking in the direction of the library.
3) Cross the corridor.
4) Enter the library.
5) Ask for the book
6) Get the book

In the above mentioned steps, actually what is important is a person, a library and a book. But these things are lost in the long procedure.

Same thing can be explained in the OOP terminology as, I went to library and got a book. The instructions are not given any importance rather they are included in the objects i.e. instructions (functions or methods) related to the book are embedded in the book object and so on.

In real life also, we give emphasis to data rather than the instructions. That's how modeling real life problems is easy in OOP.

3. Features of OOPs

All object-oriented programming languages have three traits in common: encapsulation, polymorphism and inheritance.

Encapsulation:-
Encapsulation is a mechanism that binds together code and data, and that keeps both safe from outside interference or misuse. Further, it allows the creation of an object. An object is a logical entity that encapsulates both data and the code that manipulates that data. When you define an object, you are implicitly creating a new data type.

Polymorphism:-
In literal sense, polymorphism means the quality of having more than one form. In the context of OOP, polymorphism refers to the fact that a single operation can have different behavior in different objects. In other words, different objects react differently to the same message.

Object-Oriented programming languages support polymorphism, which is characterized by the phase "one interface, multiple methods". Polymorphism is the attribute that allows one interface to be used with a general class of actions.

Polymorphism helps reduce complexity by allowing the same interface to be used to specify a general class of actions. It is the compilers job to select the specific action (method) as it applies to each situation.

Inheritance:-
Inheritance is a process by which one object can acquire the properties of another object. This feature allows you to add refinements of your own to an existing object.

4. Encapsulation and Data Abstractions

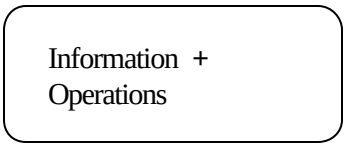
Encapsulation:-

The primitive element of object-oriented programming is an object. A object contains certain information, and knows how to perform certain operations.

Objects, which share the same behavior, are said to belong to the same class. A class is a generic specification for an arbitrary number of similar objects. You can think of a class as a template for a specific kind of objects, or as a factory, cranking out as many products as required. A class can also be treated as a user defined data type. A C++ class syntactically appears similar like structures in C.

Encapsulation is a mechanism that binds together code and data, and that keeps both safe from outside interference or misuse. Further, it allows the creation of an object. An object is a logical entity that encapsulates both data and the code that manipulates that data. When you define an object, you are implicitly creating a new data type. Encapsulation draws a capsule around related things.

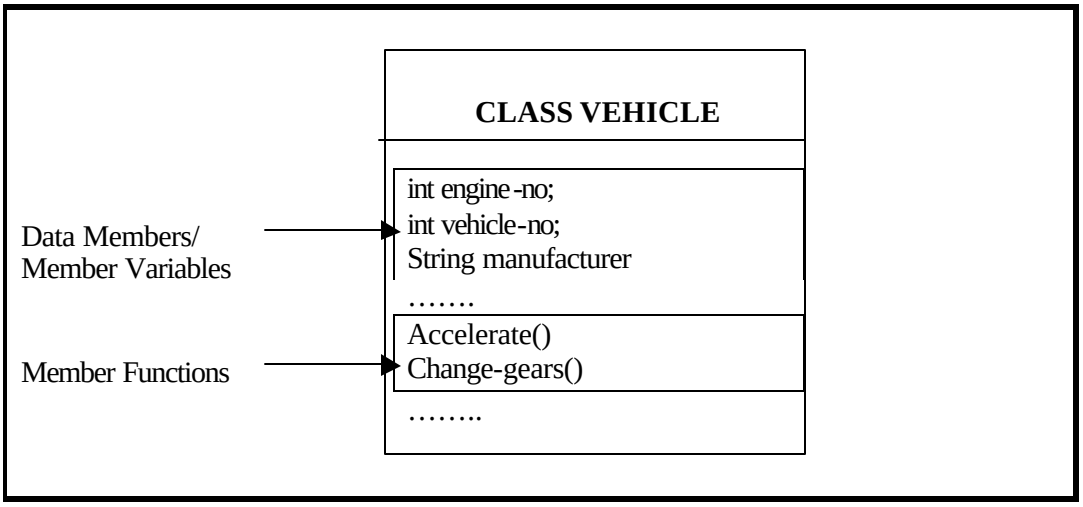
Figure 4.1 Encapsulation



Class Data Members (Information) and Members Function (Operations)

Data members are also called as properties of the object. Function members are also called as methods.

Figure 4.2 A complete class



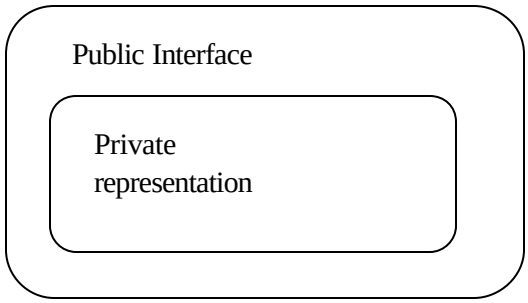
Abstraction :-

The object has one more peculiar property , data abstraction. The object has a public interface and a private representation , and keeps as information hiding. Information hiding distinguishes ability to perform some act from the specific steps taken to do so. Publicly , an object reveals its abilities: "I can do these things" , it declares, and "I can tell these things." But it does not tell how it knows or does them, nor need other objects concern themselves with that. Instead, another object requesting an operations or some information acts like a good manager. It specifies the job or asks for the information and then leaves. It doesn't hang around worrying about how the job is done or how the information is calculated.

Objects know only what operations they can request other objects to perform. This helps you take a somewhat abstract view of the object as you design it. Some details do not yet concern you , and can be deferred. You can concentrate on the essence of your design.

Another gain comes later in the lifetime of the system. You can change the internal representation of an object or implement a superior algorithm for a specific operation without changing the object's abstract, public interface.

Figure 4.3 Data Abstraction (Information Hiding)



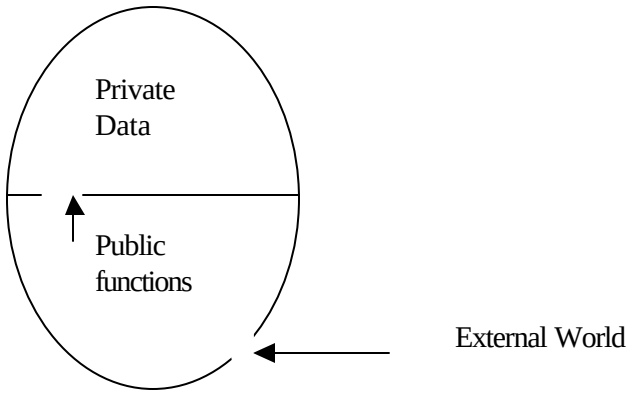
Encapsulation and information-hiding work together to isolate one part of the system from other parts, allowing code to be modified and extended, and bugs to be fixed, without the risk of introducing unnecessary and unintended side-effects. Objects are a way of putting these two principles to practical use in a program.

1. First you abstract out the functionality and information that are related, that belong together, and encapsulate them in one object.
2. Then you decide what functionality and information other objects will require of that objects. The rest you hide. You design a public interface (the outside of the capsule) that allows other objects to access what they require. The private representation (the inside of the capsule) is by default protected from access by other objects.

How can I relate these concepts with real life examples. Data abstraction means the interface is available to the external world for the access, but internal details are hidden. We all ride or drive different kind of vehicles. To operate the vehicle we are given different parts like brake, handle, accelarator etc. These parts are nothing but the interfaces. We use the vehicle with the help of the interfaces, without knowing the internal structure of the machine (data abstraction!). Now the internal functioning of brake or any other part can be changed without changing the external appearance. So

the driver is still comfortable using the brake as he was before , but at the same time the quality of the brake is improved with changed internal structure.

Figure 4.4 Encapsulation and Abstraction



From the Figure 4.4 , what we can say, the private data members and the function members are part of a class (encapsulation). The private data members are accessable only to the function members of the class. The external world has an access to the public member functions in the class.

Now lets have a detail look at the class and it's members.
Evolution of Classes
Given the amount of conceptual power embodied in the class concept, it is interesting to note that the syntax remains fairly straightforward. A class in C++,is an extension of a C structure. Basically a class allows you to create a structure, and then permanently bind all related functions to that structure. This process is known as encapsulation. It is a very simple concept, but it is the heart of object-oriented programming:
data + functions = object.

Almost all computer languages have built-in data types. For instance, a data type int means integer is predefined in C. You can declare as many variables of type int as you need in your program:

```
int day;
int count;
int x,y;
```

Similarly a class now will be used as user defined data type, and you can create as many objects of that class.

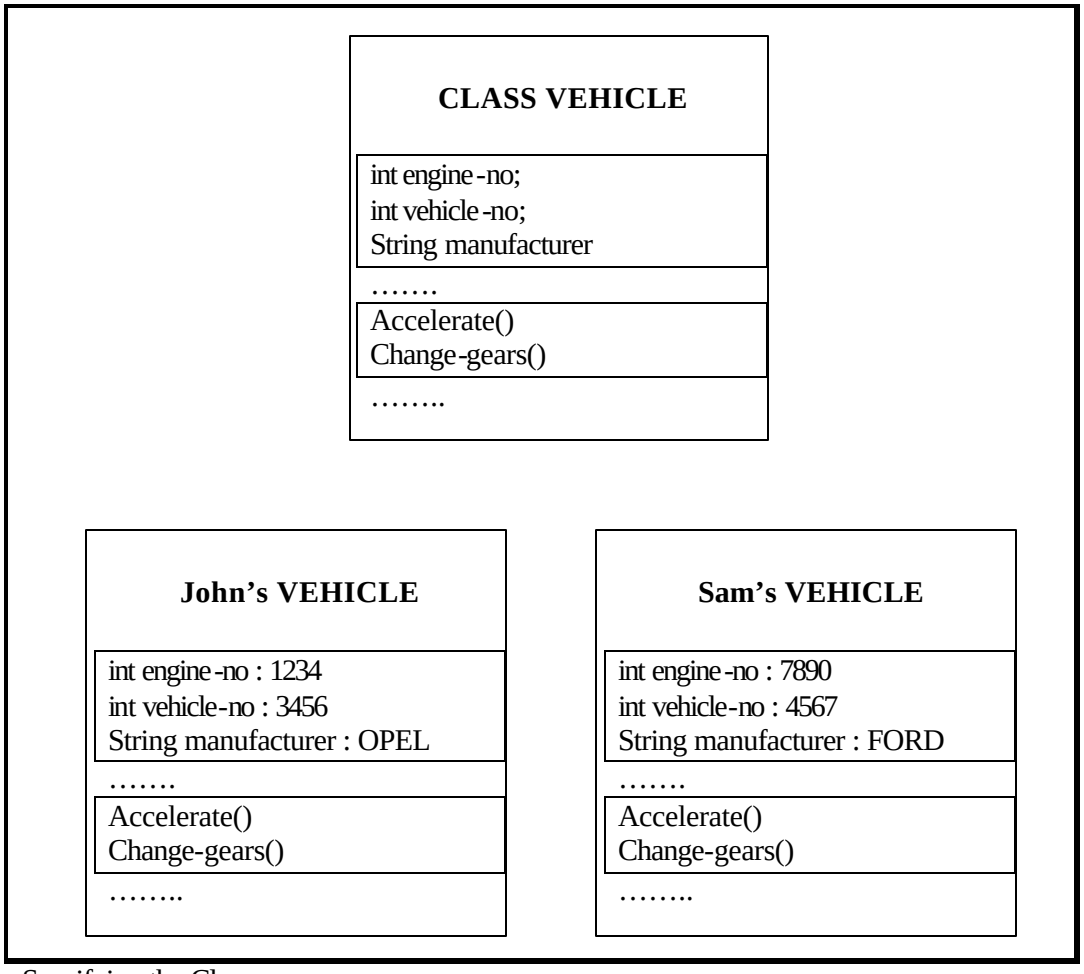
```
vehicle john-car;
vehicle sam-car;
```

It means a variable to the data type is same as an object to the class.

Here class serves as a plan, or template. It specifies what data and what functions will be included in objects of that class. Defining the class does not create any objects, just as mere existence of a type int does not create any variable. Hence we have 2 object of class vehicle, having the attributes like engine-no, vehicle-no, manufacturer and functions like accelerate() and change-gears().

Observe here that the properties and functions of a class vehicle are automatically become a part of object, but the value that each property holds for two different object is different. For example, Opel manufactured John's car. while Ford manufactures Sam's car, but both objects have a property called as manufacturer.

Figure 4.5 Actual representation of class and object



Specifying the Class

All said and done about what is a class and what is an object, let us see the basic syntax for writing the class.

The general format for writing the classes is:

```
class <class-name> {
        data-type variable1;
        data-type variable2;
        data-type variable3;

        data-type <function-name> ( function arguments) {
        }

        data-type <function-name> ( function arguments) {
        }

}
```

Note: This is a generalized format to write a class. The actual syntax might differ a little bit from language to language.

So let us write concrete definition of class Vehicle that we have been representing diagrammatically until now.

```
class Vehicle {

        int engineNo;
        int vehicleNo;
        String manufacturer;

        //function Accelerate
        public void Accelerate () {
                //some code here
        }

        //function change-gears
        public void Change-gears() {
                        //some code here
        }
} // class ends.
```

Class Data

The vehicle class contains three data members: engineNo, vehicleNo of data type int and manufacturer of data type String. There can be any number of data items in a class. The data items might precede with some visibility modifier / access specifier depending upon the language used.

Some of the common visibilities supported by common OOP languages are public, private. Public visibility means , the data is available to the external world. Private visibility means , the data cannot be accessed directly. It can be accessed only by member functions. More about it would be discussed next topics.

Member Functions:-

We need member functions (generally declared as publicly visible), to access the private data.

Member functions are functions that are included within a class. (In some object-oriented languages, such as Java, member functions are called as methods) There are two member functions in vehicle class Accelerate() and Change-gears(). The member functions can also perform operations that are quite common in classes: setting and retrieving the data stored. The setVehicleNo() function accepts the value as a parameter and sets the vehicleNo variable to this value. The showVehicleNo() displays the vehicleNo. Hence the complete class definition is as follows:

```
class Vehicle {

        int engineNo;
        int vehicleNo;
        String manufacturer;

        //function accelerate
        public void Accelerate () {
                //some code here
        }

        //function change-gears
        public void Change-gears() {
                        //some code here
        }

        //sets the value of the variable to new value.
        setVehicleNo (int a) {
                vehicleNo = a;
        }

        showVehicleNo() {
                System.out.println("The Vehicle No. is:" + vehicleNo);
        }
} // class ends.
```

In an OOP class you can have member functions with the same name but having different signatures. This is called as function overloading, an example of polymorphism. While the signature of the function is considered, the return type is not to be taken into consideration i.e. the functions must have different argument list.

e.g. void area( ) , void area(int a), void area (int a, int b) can be called as overloaded functions. But int area( ), void area( ) cannot be called as overloaded functions. In OOPs you can have overloaded ordinary functions as well as overloaded member functions.

In some of the OOP languages like C++, polymorohism can be implemented using operator overloading. This concept is not supported by Java. Operator overloading allows you to manipulate objects as, basic data types. If I have a date class, containing some date, a minus operator can be overloaded to find out difference between two dates.e.g.
Date d1, d2,d3;
D3= d1 - d2;
In this example d1,d2 and d3 are date objects . d1 and d2 are subtracted to give d3.
Now without overloading the operator , such subtraction is not possible.

Using the Class
Now that the class is defined, let us discuss the possible ways to use any class or members of that class. Any class can be used only in two fashions:

- Instantiate the class and create the object
- Write a different class inheriting from that class

This chapter explains the how to create objects from the class and call its member functions while inheritance will be described in the next chapter.

Defining Objects
The first statement in main(),

```
Vehicle v_john = new Vehicle();
Vehicle v_sam = new Vehicle();
```

defines two objects, v_john and v_sam, of class vehicle. Remember that the specification for the class Vehicle does not create any objects. It only describes how they will look when they are created. Defining an object is actually defining a variable of any data type: Space is set-aside for it in memory.
Calling Member Functions

The next two statements in main() call the member function setVehicleNo():

```
v_john. setVehicleNo(3456);
v_sam. setVehicleNo(4567);
```

These statements don't look like a normal function calls. This strange syntax is used to call a member function that is associated with a specific object. Because setVehicleNo() is a member function of the Vehicle class, it must be called in connection with an object of this class. It doesn't make sense to say

```
setVehicleNo(1234);
```

by itself, because member function is always called to act on a specific object, not on class in general. Attempting to call a method like that will produce a compiler error.

Note: only an object of that class can access Member functions of the class.

5. Constructors and Destructors

Constructors

Constructors are nonstatic member functions that determine how the object of a class is created, initialized and copied. A constructor executes automatically when an object is created. Constructors are also invoked when local or temporary objects of a class are created.

The constructor can be overloaded to accommodate many different forms of initalizations for instances of the class. The execution of the contructor does not reserve memory for the instance itself. The compiler generates the code to do this, either in static memory, on the stack, or on the heap, after which control is handed over to the constructor to do the initialization of the data members.

Rules for Constructors :

- Name of Constructors should be the same as name of the class to which it belongs.
- It is declared with no return type not even void.
- It cannot be declared as static or const .
- It should have public or protected access.
- Constructors can be overloaded

When the Constructor is called :

An instance of some class comes into existence and causes the constructor function to be invoked when

- A global or static local variable of the class is defined (the constructor is called before main( )).
- An auto variable of the class is defined within a block and the location of its definition is reached.
- A temporary instance of the class needs to be created.
- An instance of the class is obtained from free memory via the new operator.
- An instance of some class is created that contains, as a data member, an instance of the class.
- An instance of some class derived from the class is created.

```
class Box {
        private:
```

```
        int width;
        int height;

        public:

        /*
        This is a constructor of class, used to initialize data members.
        Width and height will be initialized to 5 when the object is created.
        */
        Box() {
                width = 5;
                height = 5;
        }

        void showDetails() {
                cout << "Width = "  << width;
                cout << "Height = " << height;
        }
}

void main() {
        Box b1, b2;
        b1. showDetails();
        b2. showDetails();
        }
```

Example 5.1 Box Class Definition [in C++]

Constructors without arguments

The constructor in the example 5.1 is called as a default or a no argument constructor. If we not write any such constructor , the compiler would support us a default constructor. A user defined constructor would initialize the data members to some valid values. A compiler defined constructor would either set false values to the data members or they would contain garbage, depending upon the particular OOP language. You can have at the most one such constructor in your class definition. Suppose Box is the name of our class then,

Box b1; // In C++

Would create an object b1 of type Box. (Default constructor called !)

Constructors with arguments

As mentioned before, constructors can be overloaded. We can have as many constructors with arguments (of course each with a different signature). If we write even a single constructor with arguments, the compiler would not support us the default constructor .

```
        //parameterized constructors
        Box( int a) {
                width = a;
                height = a;
        }

        Box(int a, int b) {
                width = a;
                height = b;
        }

        void showDetails() {
                cout << "Width = "  << width;
                cout << "Height = " << height;
        }
}

void main() {
        Box b1, b2(5),b3(10,12);
        b1. showDetails();
        b2. showDetails();
        b3.showDetails();
}
```

Example 5.2 Parameterized constructor (C++)

Here in the examples above, we are writing three different constructors, for the box class. This kind of technique is called as function overloading. Which function to call is decided upon the kind of parameters passed while calling the function. For example, in the code above we are creating three objects of Box class, b1,b2 and b3.

Box b1; // this declaration calls the first constructor.

Box b2(5); // this declaration calls the second constructor which gets one integer value.

Box b2(5); // this declaration calls the second constructor witch expects two integer values.

The result of this program will be:

Box b1: width = 0, height= 0
Box b2: width = 5, height= 5
Box b1: width = 10, height= 12

The first constructor with no arguments, is called as a default constructor of the class. The other two constructors are called as parameterized constructors. This is a specific example of constructor overloading. The same technique is applicable to any other functions in the class.

Destructors

A destructor is a function, which is called automatically whenever an instance of the class goes out of existence. The destructor is used to release space on the heap that the instance currently has reserved.

Rules for Destructors

- Its name is the same as that of the class to which it belongs, except that the first character of the name must be a tilde(~).
- It is declared with no return type, not even void, since it cannot return a value.
- It cannot be declared static or const.
- It takes no input arguments, and so cannot be overloaded.
- It should have public access in the class declaration.

When a destructor function is called

The destructor function for a class is called :

- After the end of main( ) for all static, local to main( ), and global instances of the class.
- At the end of each block containing an auto variable of the class.
- At the end of each function containing an instance of the class as argument.
- To destroy any unnamed temporary instances of the class after their use.
- When an instance allocated on the heap is destroyed via delete.
- When an object containing an instance of the class is destroyed.
- When an object of a class derived from the class is destroyed.

Static Class Data

Having said that each object contains its own separate data, we must amend that slightly. If a data in a class is defined as static, then only one such item is created for the entire class, no matter how many objects are created from that class. Static item is useful when all the objects of the same class need to share a common item of information.

As an example, suppose an object needs to know how many other objects of its class were in the program. We can use a static data member over here to fulfill the functionality.

```
class InstanceCounter {

        static int instances;

        InstanceCounter () {
                instances++;
        }

        static void showCounterValue() {
System.out.println("The number of objects created till now are:" + instances);
        }

        public static void main(String args[]) {

                InstanceCounter.showCount(); // static methods are called
using  //class name
                InstanceCounter ic1 = new InstanceCounter();
                showCount();
                InstanceCounter ic2 = new InstanceCounter();
                InstanceCounter.showCount();
        }
}
```

Example 5.3 Static class data in Java

The output of this program will be:

```
The number of objects created till now are: 0
The number of objects created till now are: 1
The number of objects created till now are: 2
```

It can be observed that, all instances share a same copy of variable instances and increment the same variable. Apart from declaring the static variables, we can declare even functions to be static. The main advantage of declaring a static function is that, static functions can be called, even without creating an object of that class, by using the class name. For example, to print the value of the counter, we can write a static function showCounterValue() that prints the current value of the counter. As this method is declared as static, it can also be called using class name.

To take one more example, interest rate of some bank class can also be declared as static. There are two benefits of it. First , the same copy of rate of interest can be shared by all the instances (objects) of the class. Secondly, you can access , this value without having any object created for the class. Had it been the class member, you would have, required an object to access this value. It means , if tomorrow anybody comes and enquires about the rate of interest of the bank (Let's say for a newly opened branch), we would say , we cannot access the value because by now nobody has opened an account in our branch (no object is created!).

```
Class item {
        static int count ;
        int number ;
         public :


                void getdata(int a){
                number = a;
                count ++; }
                void getcount (void) {
        cout << "count: " ;
                cout << count << "\n" ; }
        } ;
        int item :: count ;
        main () {
                item a , b ;
                a.getcount(); b.getcount();
                a.getdata(); b.getdata() ;
                cout << "After reading data ";
                a.getcount(); b.getcount();
        }
```

Example 5.4 Static class data in C++

6. Types of Variables

Let us again have a look at the method addTwoBoxes() from previous example.

```
Box addTwoBoxes(Box b) {
                int x = width + b.width;
                int y = height + b.height;
                int z = depth + b.depth;
                Box newb = new Box(x, y, z);
                Return newb;
}
```

The number of variables used by this particular function are b, x, y, width, height and so on. These all variables are bifurcated into two categories:

- Local Variables
- Instance Variables

Local Variable

The variables declared within a function or method or any block of code and the arguments passed to a function are called as local variables. The scope of all the local variables is with that block. It simply means that, the local variables are not accessible outside the block, in which they are defined.

```
Box addTwoBoxes(Box b) {
/*
  width, height, depth : instance variables
  x, y, z, b, newb        : local variables
*/
                int x = width + b.width;
                int y = height + b.height;
                int z = depth + b.depth;
                Box newb = new Box(x, y, z);
                Return newb;
}
int volume() {
        return x*y*z;  // will produce a an error as x, y and z are not available.
}
```

Hence the variables, x,y and z of type int, b and newb of type Box, all will come under the category of local variables.

Note: The scope of local variables is limited to a block.

Instance Variables

The non-static variables declared as a part of class are called as instance variables. The instance variables are accessible by all the functions written within the class. Hence instance variables can be thought of as global variables within the boundaries of the class.

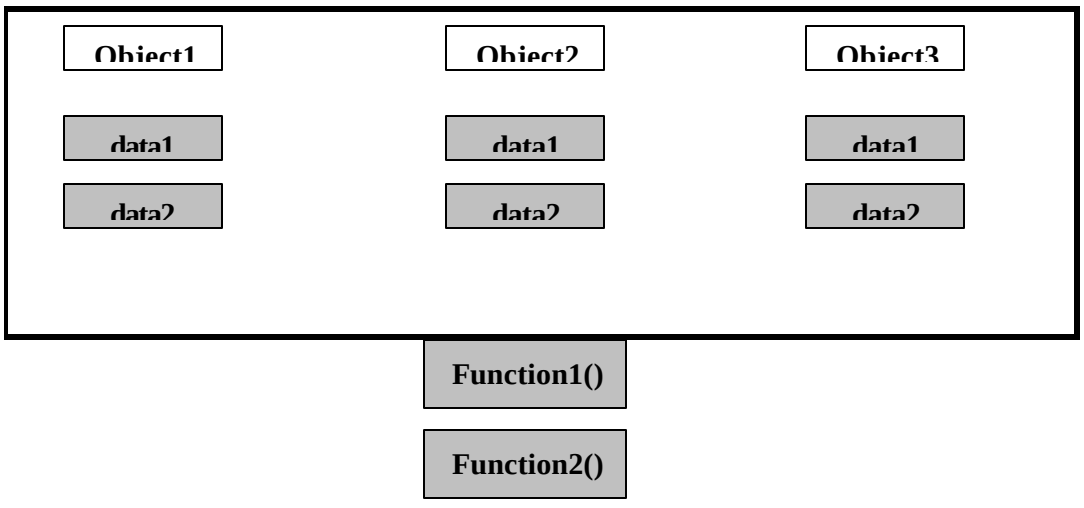
The variables, width, height, and depth accessed in the previous snippet are instance variables. There is one more category of variables called as static variables, which will be discussed in the following sections.

Classes, objects and memory

It probably is the impression that each object created from a class contains separate copies of that class data and member functions. This is a good first approximation, since it emphasizes that objects are complete, self-contained entities, designed using the class specifier. The mental image here is o cars(objects) rolling off an assembly line, each one made according to the blue print(class).

However things are not quite so simple. It's true that each object has its own separate data items, all objects in a given class share the same member function. The member functions are created and placed in memory only once – when they are defined in the class specifier. This makes sense; there is rally no point in duplicating all member functions in a class every time you create a new object another object of that class, since the functions for each object are identical. The data items, however, will hold different values, so there must be separate instance of each data item in each object. The same thing is pictorially represented in the following diagram.

Figure 6.1 Multiple Objects sharing function definition



Points to remember:-

- OOP's talk in terms of objects
- Objects are instances of classes
- A class is nothing but a user defined data type having data members and function members, this feature is called as encapsulation
- You can imagine a class to be a blue print, used for constructing objects.
- Data members can be made accessible only to the function members, thus hidden from external world. This feature is called as data abstraction.
- Objects are created as we otherwise create ordinary variables
- A different set of data members is created for each object wherein a single set of function members is referred by all the objects
- Constructors are the member functions having the same name as the class
- Constructors can be used to initialize the data members of the objects
- Constructors can be default (no argument) or parameterized in nature
- We can write as many parameterized constructors, each having different parameter list
- If we do not write any constructor compiler would give us default constructor
- Compiler would not give us default constructor even if we write a single parameterized constructor
- Constructor cannot return value (not even void)
- Destructor is also a member function having the same name as the class, followed by a special character tilde(~) in C++ (JAVA classes do not have destructors)
- There can be only one destructor in a class
- Destructor neither take parameter nor return value
- Unlike ordinary data members single copy of static data members is shared by all the objects.
- Static data members can be used without having created even a single object
- Variables as well as methods can be declared static.
- Static data can be accessed using class name.

7. Polymorphism

In literal sense, poly means many and polymorphism means many forms and this is the quality of having more than one form.

In the context of OOP, polymorphism is a concept wherein a name can refer to an object of any of multiple types (classes), as long as the types are all related via a common super class and the name is declared to be able to refer to an object created from the common super class.

By using polymorphism, software developers can write program code that performs common operations on related types of objects, without requiring the software developers to be concerned about what specific types of objects the program code will work with. In other words, by using polymorphism, software developers can write program code to invoke common operations (methods) on a variety of object types. For example, have a look at the example

```
void  repchar();
void repchar(char);
void repchar(char, int);
void main()
{
        repchar();
        repchar('=');
        repchar('+', 30);
}

void repchar()
{
        for(int j = 0; j < 45; j++)
                cout << '*';
}
void repchar(char c)
{
        for(int j = 0; j < 45; j++)
                cout << c;
}
void repchar(char c, int n)
{
        for(int j = 0; j < n; j++)
                cout << c;
}
```

Example 7.1 Function overloading in C++

repchar() is function , overloaded with different argument list . Depending upon the parameters passed , appropriate version of the function would be called.
A common operation (method) invoked via polymorphism may be specially defined for each object type (class), as long as the method for each object type (class) uses the same number and types of parameters and has the same type of return value.

Polymorphism is implemented using two basic techniques in OOP:

- Function Overloading
- Function Overriding

The implementation of both these techniques is discussed earlier. As you get used to using these techniques, you will understand the importance of polymorphism and impact of it on object-oriented analysis and design (OOAD). As of now it is sufficient to know that there are two categories of polymorphism:

- Static Polymorphism also called as compile time Polymorphism
- Dynamic Polymorphism also called as run time overloading

Static Polymorphism is implemented by function overloading or operator overloading while function overriding supports dynamic polymorphism.

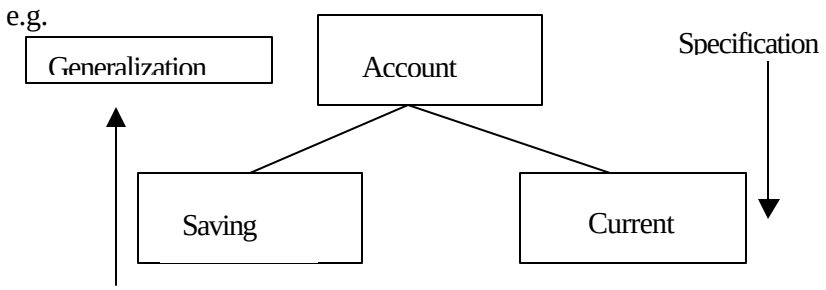
8. Inheritance

Inheritance is probably the most powerful feature of object-oriented programming, after classes themselves. Inheritance is the process of creating new classes, called derived classes, from existing or base classes. The derived class inherits all the capabilities of the base class but can add some specificity of its own. The base class is unchanged by this process.

Let's say that you have a Box class, and now you want to modify it. In the old world of programming you would take the source and start changing things. In the object oriented world of programming you do things differently. What you do instead is leave the existing class alone and then layer your changes on top of it using a process called inheritance. Layering through inheritance lies at the very heart of object oriented programming. It is a totally different way of doing things, but it has several important advantages:

- Let's say that you bought the Box class from someone else, so you don't have the source code. By leaving the existing class alone and layering your changes on top of it you don't need to have the source.
- The existing class is completely debugged and tested. If you modify its source, it has to go through the testing process again to be re-certified. Changes you make might also have side effects that aren't detected immediately. By layering the changes on top of the existing class, the existing class never changes and therefore remains bug-free. Only the new pieces must be tested.
- The layering process forces you to think in a generic-to-specific way. You create a generic class like a Box, and then layer specificity on top of it. A nice bonus of this way of thinking is that the generic classes are useful in many different programs. A Box, for example, is useful in a lot of places. Each new program layers its own specifics onto the generic list, but the generic box stays the same everywhere.

Figure 8.1



In the above mentioned example, account is called as base class and saving and current are called as derived classes.

- If the "base class" is improved, all classes built on top of it take advantage of those improvements without modification. For example, say that the list class is changed so that it sorts 10 times faster than it used to. Now every class built on top of the list class sorts 10 times faster as well, without modifying anything.

Most importantly it permits code reusability. Once the base class is written and debugged, it need not be touched again but can nevertheless be adapted to work in different situations. Reusing existing code saves time and money and increases the program reliability.

One result of reusability is the ease of distributing class libraries. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

Derived and Base Classes

Let us carry forward the same example that we have been using through out the previous chapter. Suppose we are box-manufacturing company and manufacture various types of boxes. Some of the boxes are metal while others are cardboard boxes. Now for a paper box, the weight of the box is very negligible, but a metal box weight is considerable. Box class can still specify objects of paper boxes, but for metal box we will write one more class deriving from Box class.

```
class Box {

        int width,height,depth;

        Box()  {
                System.out.println("In Base Class");
        }

        Box(int a, int b, int c)  {
                width=a;
                 height = b;
                depth=c;
        }

        Box( int a)  {
                 width = height = depth = a;
        }
        void show() {
            System.out.println("The details of  this box are:");
            System.out.println("Width = " +width);
            System.out.println("Height = " +height);
            System.out.println("Depth = " +depth);
            System.out.println("\n\n");
        }

        int volume() {
                return width*height*depth;
        }

}

// MetalBox class inheriting from Box class.

class MetalBox extends Box {  // line 1

        int weight; //specific property weight added to the derived class

        MetalBox() {
```

```
        // variables of base class are directly accessible from the derived class
by virtue of inheritance.
                System.out.println("In Derived Class");
                width = 10;
                height = 10;
                depth = 10;
                weight = 10;
        }

        public static void main(String args[]) {

                MetalBox mb = new MetalBox();
                int x = mb.volume();
                System.out.println("Volume : " + x);

        }
}
```

Example 8.1 Establishing inheritance relationship in Java

```
class Box {
        Private:
                int width;
                int height;
                int depth;
        Public:
        Box()  {
                System.out.println("In Base Class");
        }

        Box(int a, int b, int c)  {
                width=a;
                 height = b;
                depth=c;
        }
        Box( int a)  {
                 width = height = depth = a;
        }
        void show() {
             cout << "The details of  this box are:" ;
            cout << "Width = " << width ;
            cout << "Height = " << height;
            cout << "Depth = " << depth;
            cout << "\n\n";
        }

        int volume() {
```

```
        return width * height * depth;
        }
}

// MetalBox class inheriting from Box class.

class MetalBox : public Box {  // line 1
        int weight; //specific property weight added to the derived class

        MetalBox() {
                // variables of base class are directly accessible
                        from the derived class by virtue of inheritance.
                System.out.println("In Derived Class");
                width = 10;
                height = 10;
                depth = 10;
                weight = 10;
        }
}
void main() {
        MetalBox mb ;
                int x = mb.volume();
                cout << "Volume : " << x ;
}
```

Example 8.2 Establishing inheritance relationship in C++

Here in the examples above, Box is a base class, MetalBox is derived class. The line marked as line 1 is the syntax for specifying that Box is a base class for MetalBox class. Observe one more thing over here, MetalBox has one constructor, that accesses the variables such width, height and depth declared in the class Box directly. It means, everything declared in the base class is also made a part of sub class by virtue of inheritance. Because of the same reasons we could call the volume() function directly without writing a volume() function in the derived class.

The output of this program will be

```
In Base Class
In Derived Class
Volume : 1000
```

A very peculiar thing to observe here is the order in which the constructors are called. The output shows that even before calling the first statement in MetalBox constructor, the Box class default constructor is called and then Metal Box's default constructor gets executed.

Note: The Constructors are called from top to bottom in that sequence.

Visibility Mode:-

Visibilty mode in inheritance decides the way , in which the class members can be inherited by the derived class.

The visibility mode can be private , public, or protected. Private and protected members in a class are treated equally i.e. both available only to the member functions or methods and not to the external world. If they are same then what do we use protected mode for? Protected mode is used to be inherited by the derived class.Lets' see it. A meber declared as protected is accessable by the meber function within its class and any class immediately derived from it. It cannot be accessed by the function outside these two classes.

The derived class can inherit the base class using either of the three ways, i.e. private, public or protected. The visibility mode in heritance decides, what data of base class derived class can access.

Lets see the rules for visibility mode in C++

1) private data of the base class is never available to the derived class
2) protected data of the base class inherited publicly, becomes protected in derived class
3) Protected data inherited privately , becomes private in the derived class
4) Protected data in the base class inherited protectedly becomes protected in the derived class
5) Public data in base class derived privately becomes private in the derived class
6) Public data from base class derived protectedly becomes protected in the derived class
7) Public data in the base class inherited publically, becomes public in the derived class.

In short, what we can say :-

Base class visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

In Java , all classes are derived publically only.

Example :- Public Inheritance

```
//Simple inheritance with public visibility

#include <iostream.h>
class parent
{
        int x;
public:
        int y;
```

```
void get_data(void);
        int get_x(void);

        void disp_xdata(void);

};

class child :public parent
{
        int z;
public:
        void add(void);
        void result(void);

};

void parent::get_data(void)
{
        x=10;
        y=20;
}

int parent::get_x(void)
{
        return x;
}
void parent::disp_xdata(void)
{
        cout <<x;
        cout<<endl;
}

void child ::add(void)
{
        z=y+get_x();
}
void child::result(void)
{
        cout<<get_x()<<endl;
        cout<<y<<endl;
        cout<<z;
}

void main()
{
        child ch1;
        ch1.get_data();
```

```
        ch1.add();
        ch1.result();
        ch1.y=30;
        ch1.add();
        ch1.result();
}
```

Example 8.3 Inheritance with Public visibility in C++

Exmaple of Inheritance with private visibility

//Simple inheritance with private visibility

```
#include <iostream.h>

class parent
{
        int x;
public:
        int y;
        void get_data(void);
        int get_x(void);
        void disp_xdata(void);

};
class child :private parent
{
        int z;
public:
        void add(void);
        void result(void);


};

void parent::get_data(void)
{
        x=10;
        y=20;
}
int parent::get_x(void)
{
        return x;
```

```
}
void parent::disp_xdata(void)
{
        cout <<x;
        cout<<endl;
}

void child ::add(void)
{
                get_data();
        z=y+get_x();
}
void child::result(void)
{
        disp_xdata();


cout<<y<<endl;

        cout<<z;
}


void main()
{
        child ch1;
        //ch1.get_data();
        ch1.add();
        ch1.result();

        //ch1.y=30;
        ch1.add();
        ch1.result();

}
```

Example 8.4 Inheritance with Private visibility in C++

Types of Inheritance:-

1) Single level:-

As the name suggests, in this type of inheritance only one level of the inheritance is there, i.e there will be one base class and only one derived class.

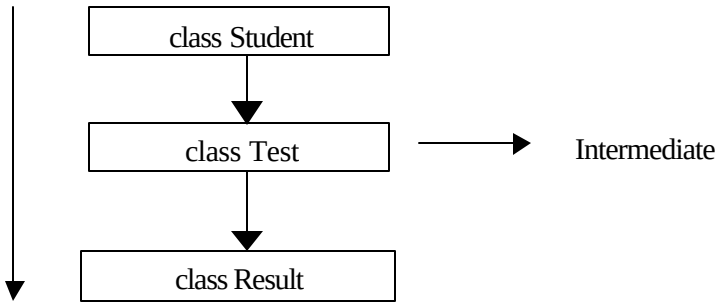
Figure 8.2

2) Multilevel Inheritance:-

Inheritance flows down in more than one level.

Figure 8.3

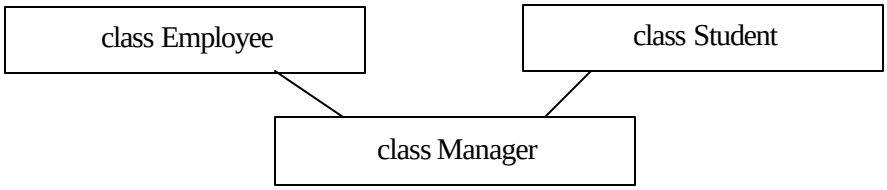
Inheritance Path



3) Multiple Inheritance:-

Inheritance with more than one base class.

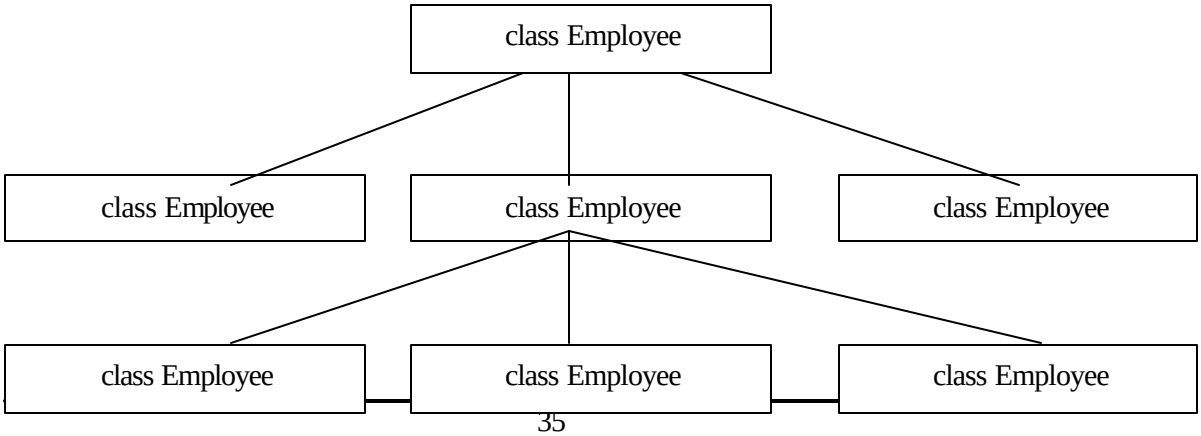
Figure 8.4



4) Hierarchical Inheritance:-

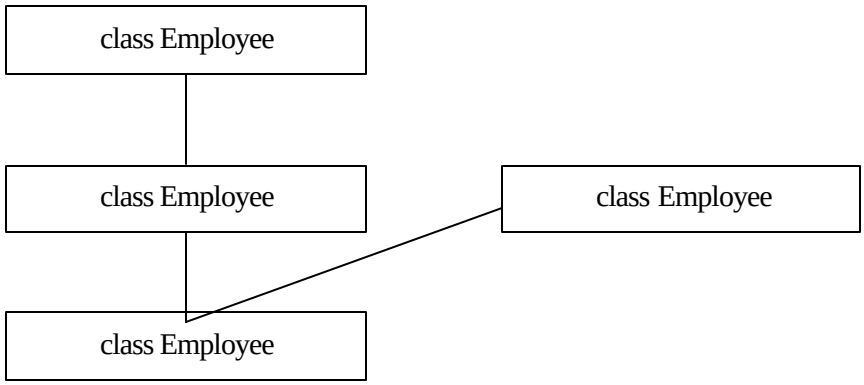
Inheritance like tree structure.

Figure 8.5



5) Hybrid Inheritance:-
Two or more different ways of inheritance are combined together.

Figure 8.6



Constructors in derived class

The derived class constructors also will follow all the rules applied to constructors. We can also overload the constructors as we did it for box class. But the potential glitch in the above program is when we want to call other constructor than the default one. As we saw in the previous program, if no constructor from base class is called compiler will substitute it with a call with default constructor. As you can see in the previous program, the statements for initialization of width, height and depth are repeated in Box class as well as in MetalBox class. This could be avoided if we have some way to call the base class constructors from the child class. The following example demonstrates the same thing.

```
class Box {

        int width,height,depth;

        Box()  {
                System.out.println("In Base Class");
        }

        Box(int a, int b, int c)  {
                width=a;
                 height = b;
                depth=c;
```

```
        }

        Box( int a)  {
                width = height = depth = a;
        }

        int volume() {
                return width*height*depth;
        }

        void show() {
            System.out.println("The details of  this box are:");
            System.out.println("Width = " +width);
            System.out.println("Height = " +height);
            System.out.println("Depth = " +depth);
            System.out.println("\n\n");
        }
}

// MetalBox class inheriting from Box class.

class MetalBox extends Box {  // line 1

        int weight; //specific property weight added to the derived class

        MetalBox() {

        // variables of base class are directly accessible from the derived class by virtue of inheritance.
                super (10); // will replace the following statements from previous //example width = 10;  height = 10; depth = 10;
                System.out.println("In Derived Class");
                weight = 10;
        }

        public static void main(String args[]) {

                MetalBox mb = new MetalBox();
                int x = mb.volume();
                System.out.println("Volume : " + x);

        }
}
```

Example 8.5 Derived Class Constructor(In Java)

In the similar fashion we can call up all the constructors, provided that they exist and we are passing it with proper parameters.

Note: If no super call exists, then the compiler will automatically include a call to a default constructor of the parent class.

Let's see one more example of constructors in derived class:-

```
class base
    {
        protected :
            int num;
        public :
            base ( int n = 0) : num (n)  { }
            int get_number( ) const
            {

            return num;
        }
        };
        cla ss derived : public base
        {
            int num;
        public :
            derived(int n = 1) : num (n)  { }//Derived class constructor
            int get_number( ) const
            {

                        return num + 1;


        }
        };

        void main( )
        {
            derived d;
```

```
        // Calls derived::get_number( )
                cout << d.get_number( );
        // Calls base::get_number( )
        cout << d.base::get_number( );
}
```

Example 8.6 Derived class constructor (C++)

The constructor of the base class is always called first. If do not initialize the derived class constructor with some value, then the default constructor of the base class would be called.

Points to remember:-

- Polymorphism means the ability to take more than one form.
- Polymorphism can be achieved using function overloading or operator overloading (not supported by JAVA)
- Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface.
- In function overloading you can have many functions of same name, and different signature. While considering the signature, the return type of the function is not considered. The argument list of all overloaded functions must differ.
- Inheritance is another important feature that allows, to create a class, having some some members of it's own and some members inherited from other existing class(es)
- Inheritance can be of many types:-
    - Single Inheritance
    - Multilevel Inheritance
    - Multiple Inheritance
    - Hirarchical Inheritance
    - Hybrid Inheritance

    All these types are supported by C++, Java supports only single and multilevel inheritance
- Advantages of inheritance:-
    - Generic-to-specific way of thinking.
    - Code reusability.
    - Fully tested and debugged code is used.
    - Increases program reliability.
    - Saves time and money.
    - Distribution of class libraries is easy.
    - Source code is not required.
    - It does not touch the prewritten code.

9. Polymorphism in Inheritance

Overriding Member Functions

The ability of a subclass to override a method in its super class allows a class to inherit from a super class whose behavior is "close enough" and then override methods as needed.

You can use member functions in a derived class that have the same name as those in the base class. The basic reasons behind doing this is to override the functionality implemented by the parent class and write the different functionality in the child class.

In the following example, there is show() method in the Box class that shows only width, height and depth of a box; while an object of MetalBox class should have a show method that not only gives the details of width, height and depth but also of the weight attribute that's added in the MetalBox class. The following example shows that:

```
class Box {

        int width,height,depth;

        Box()  {
                System.out.println("In Base Class");
        }

        Box(int a, int b, int c)  {
                width=a;
                 height = b;
                depth=c;
        }

        Box( int a)  {
                 width = height = depth = a;
        }

        int volume() {
                return width*height*depth;
        }
        void show() {
            System.out.println("The details of  this box are:");
            System.out.println("Width = " +width);
            System.out.println("Height = " +height);
```

```
            System.out.println("Depth = " +depth);
              System.out.println("\n\n");
        }
}

// MetalBox class inheriting from Box class.
class MetalBox extends Box {  // line 1

        int weight; //specific property weight added to the derived class

        MetalBox() {

                super (2,3,4);  // calls the other constructor expecting 3 integers.
                System.out.println("In Derived Class");
                weight = 10;
        }

        void show() {

                  super.show();
                   System.out.println("Weight = " +weight);
        }

        public static void main(String args[]) {

                  MetalBox mb = new MetalBox();
                  int x = mb.volume();
                  mb.show();
                  System.out.println("Volume : " + x);

        }
}
```

Example 9.1 Function Overriding in Java

In the example above, we have a method show() in Box class as well as in MetalBox class. Both the methods are having the same signatures. This kind of programming practice is called as function overriding. Also there is one interesting thing to be observed, that from the sub-class, we make use of super.show() construct to call the base class's show method.

Which Function is Used

The MetalBox class contains a show() function. This function has the same name and the same argument and return types, as the function in Box class. When we call these functions from main, in statements like

```
mb.show();
```

how does the compiler know, which of the two show() functions to use? Here's is the rule: When the same function exists in both the main as well as derived classes the function from the derived class is executed. (This is true if the object is of derived class. For objects of base class, the function from base class will be called as the base class does not know anything about derived class.) Hence the output of the above call will be,

```
The details of  this box are:
Width = 2
Height = 3
Depth = 4

Weight = 10
Volume = 24
```

In C++ also, you can use functions in a derived class that have the same name as those in the base class. You might want to do this so that calls in your program work in the same way for objects of both base and derived classes and to add more functionality.
If an instance or member function of the derived class refers to a data member of the base class that has been overridden, the member from the derived class will be used by default. To refer to the member function of the base class that has been overridden in the derived class, the base class name and the scope resolution operator are required. ie. base-classname::member-function-name.

```
class base
{
protected :
        int num;
public :
        base ( int n = 0) : num (n)  { }
        int get_number( ) const
        {
```

```
                return num;
        }
};
class derived : public base
{
        int num;
        public :
        derived(int n = 1) : num (n)  { }

int get_number( ) const
        {
                return num + 1;
        }
};

void main( )
        {
        derived d;
        // Calls derived::get_number( )
                cout << d.get_number( );
        // Calls base::get_number( )
        cout << d.base::get_number( );
        }
```

Example 9.2 Overriding Member functions in C++

The output is

```
2
0
```

Virtual Function:-

In case of overriding functions, the object of some type i.e. either the base or derived type would call appropriate function. But it is very difficult to decide the caller object dynamically in run time environment. In such situations we can use the virtual key word. This keyword can identify caller object and a call will be given to respective function. Generally the function is made virtual in the base class.

Lets observe following codes

1)

```
class  employee
{
        public :
                int empno;
        public :
                employee(int n)
                {       empno = n;      }

                void display(void)
                {
                        cout <<  empno ;
                 }
};class  professor : public employee
{
        public :
                        int exp;
        public :
                professor(int  e)
                {

                        exp = e;
                }
                void display(void)
                {
                                cout << "Emp No    : = " << empno ;
                                cout << "Experience : = " << exp ;
                }
};
void main(void)
 {
        employee  emp(555);
        professor pro(2);
        employee* e=&emp;
        e->display(); // calls base class function
        e=&pro;
        e->display(); // also calls base class function
}
```

Example 9.3 Inheritance & overridden function without virtual keyword

2) Now we will make the display function virtual.

```
class  employee
{
        public :
                int empno;
        public :
                employee(int n)
                {       empno = n;      }

        virtual void display(void)
        {
                cout <<  empno ;
         }
};class  professor : public employee
{
        public :
        int exp;
        public :
                professor(int  e)
        {
                exp = e;
        }
       void display(void)
        {
        cout << "Emp No    : = " << empno ;
        cout << "Experience : = " << exp ;
}
};
void main(void)
{
        employee  emp(555);
        professor pro(2);
        employee* e=&emp;
        e->display(); // calls base class function
        e=&pro;
        e->display(); // calls derived  class function
}
```

Example 9.4 Inheritance , overridden functions with virtual keyword

You can observer the change in the output. In example 1) e->display was always calling display() of base class, because e is a pointer of base type;
In example 2) e->display( ) is calling display of either base or derived class , depending upon what e is pointing to.

In Java you don't have to write the keyword virtual. By default the overriding will take care of calling proper function.

10. Abstract classes & Functions

Now that we have already discussed, concept of inheritance and multiple inheritance, let us start with abstract functions and classes.

Sometimes, a class that you define represents an abstract concept and, as such, should not be instantiated. Take, for example, food in the real world. Have you ever seen an instance of food? No. What you see instead are instances of carrot, apple, and (our favorite) chocolate. Food represents the abstract concept of things that we all can eat. It doesn't make sense for an instance of food to exist.

Similarly in object-oriented programming, you may want to model an abstract concept without being able to create an instance of it. For example, the Employee class represents the abstract concept of employee. It makes sense to model employees in a program, but it doesn't make sense to create a generic employee object. Instead, the Employee class makes sense only as a super class to classes like Laborer and Manager, both of which implement specific kinds of employees. A class such as Employee, which represents an abstract concept and should not be instantiated, is called an abstract class. An abstract class is a class that can only be sub classed-- it cannot be instantiated.

Also , CalculateSalary() method in the base class Employee is always “dummy” method. If this method is ever called, you’ve done something wrong. The reason is because the implementation of CalculateSalary() method cannot be defined in the Employee class as the basic rule for calculating he employee salary will differ from employee to employee depending upon whether it’s a Laborer kind of employee or Manager kind of employee. That’s also because the intent of Employee is to create a common interface for all the classes derived from it.

The only reason to establish this common interface is so it can be expressed differently for each different subtype. It establishes a basic form, so you can say what’s in common with all the derived classes. Another way of saying this is to call Employee an abstract base class (or simply an abstract class). You create an abstract class when you want to manipulate a set of classes through this common interface. All derived-class methods that match the signature of the base-class declaration will be called using the dynamic binding mechanism. If you have an abstract class like Employee, objects of that class almost always have no meaning. That is, Employee is meant to express only the interface, and not a particular implementation, so creating an Employee object makes no sense, and you’ll probably want to prevent the user from doing it.

Java provides a mechanism for doing this called the abstract method. This is a method that is incomplete; it has only a declaration and no method body. Here is the syntax for an abstract method declaration:

```
abstract void CalculateSalary();
```

A class containing abstract methods is called an abstract class. If a class contains one or more abstract methods, the class must be qualified as abstract. (Otherwise, the compiler gives you an error message.)

If you inherit from an abstract class and you want to make objects of the new type, you must provide method definitions for all the abstract methods in the base class. If you don't (and you may choose not to), then the derived class is also abstract and the compiler will force you to qualify that class with the abstract keyword.

It's possible to declare a class as abstract without including any abstract methods. This is useful when you've got a class in which it doesn't make sense to have any abstract methods, and yet you want to prevent any instances of that class.

Here's an example showing the use abstract classes and methods:

```
abstract Class Employee {

        double total-salary;

        Employee() {
        }

        // This is the implementation for abstract method.
        abstract void CalculateSalary(int a, int b) ;

        void showSalaryDetails() {
                System.out.println("Salary : " + salary);
        }
}

class Laborer extends Employee{

        Laborer () {
                super();
                System.out.println("Laborer Employee");
        }

//concrete implementation of draw in the sub class
        void CalculateSalary(int noOfHours, int ratePerHour)  {
                salary = ratePerHour * noOfHours;
        }
}

class Manager extends Employee {

        Manager () {
                super();
                System.out.println("Manager Employee");
        }
```

```
//concrete implementation of draw in the sub class
        void CalculateSalary(int hra , int da )  {
                double incentives = da * 0.1* 30 ;
                salary = hra + da + incentives.
        }
}

class Implementation {

        public static void main(String args[]) {

        //These statements will print the salary of laborer kind of employees.
                Employee e = new Laborer ();
                e.CalculateSalary(40, 25); // line 1
                e.showSalaryDetails();

        // These statements will print the salary of manager kind of employees.
                e = new Manager();
                e.CalculateSalary(2000, 500); // line 2
                e.showSalaryDetails();

        /*
        This statement will be wrong cause abstract classes cannot be
instantiated and hence we cannot create an object of Employee class.
        */

        //      e = new Employee() ;
        }
}
```

Example 10.1 Abstract Classes & Polymorphism in Java

The actual result of the line 1 and line 2 is different because, line 1 calls the method from class Laborer while line 2 calls the method from Manager class.

The output of this program will be:

```
Laborer Employee
Salary : 1000
Manager Employee
Salary : 4000
```

Note: It is not necessary for an abstract class to have an abstract method; but a class having at least one abstract method must be declared abstract.

In C++ , you can make a class abstract by making a virtual function as pure virtual function. To make a function a pure virtual function , you have to equate it to zero and of course it is not implemented in the base class.

Let's visit our employee example once again , to learn the concept of abstract classes.

```
class  employee
{
        public :
                int empno;
        public :
                employee(int n)
                {       empno = n;      }
        void display(void)=0
        };class  professor : public employee
{
        public :
int exp;
        public :
                professor(int  e)
                {
                        exp = e;
                }
                        void display(void)
                {
                                cout << "Emp No    : = " << empno ;
                                cout << "Experience : = " << exp ;
                }
};
void main(void)
 {
employee  emp(555); /* compiler would give an error ! instance of an abstract
class cannot be created*/
        professor pro(2);
        employee* e;
        e=&pro;
        e->display(); // also calls base class function
}
```

Example 10.2 Abstarct Classes inC++

Points to remember:-

- Polymorphism achieved using operator overloading or function overloading is called as compile time polymorphism
- Polymorphism achieved using function overriding is called as run-time polymorphism
- In function overloading, name of the function is same but signature (return type of the function not to be considered) must be different .
- In function overriding , the name as well as the signature of the function must be exactly same.

- Function overloading is within a class
- Function overriding is across the classes, used in inheritance.
- Calling of appropriate functions for a proper object becomes possible in run time environment using virtual key word (in C++). In Java it is managed naturally.
- An abstract class is the class for which instance (object) cannot be created.
- Abstract classes are used for the purpose of inheritance.
- In C++ a class is made abstract by making a virtual function as pure virtual function (by equating it to zero and not implementing it)
- In Java the keyword abstract would make the class as an abstract class