UNIT – 5

Data Abstraction and Object Orientation

- Object oriented programming
- Encapsulation and Inheritance
- Initialization and finalization;
- Dynamic method binding
- Multiple inheritances
- Object oriented programming revisited

UNIT 5

Object oriented programming

Object-oriented programming (**OOP**) is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, which are instances of classes, are used to interact with one another to design applications and computer programs.

An object-oriented program may be viewed as a collection of interacting objects, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent "machine" with a distinct role or responsibility. The actions (or "methods") on these objects are closely associated with the object. For example, OOP data structures tend to "carry their own operators around with them" (or at least "inherit" them from a similar object or class) - except when they have to be serialized.

Simple, non-OOP programs may be one "long" list of statements (or commands). More complex programs will often group smaller sections of these statements into functions or subroutines each of which might perform a particular task. With designs of this sort, it is common for some of the program's data to be 'global', i.e. accessible from any part of the program. As programs grow in size, allowing any function to modify any piece of data means that bugs can have wide-reaching effects.

In contrast, the object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the program. Instead, the data is accessed by calling specially written functions, commonly called methods, which are bundled in with the data. These act as the intermediaries for retrieving or modifying the data they control. The programming construct that combines data with a set of methods for accessing and managing those data is called an object. The practice of using subroutines to examine or modify certain kinds of data was also used in non-OOP modular programming, well before the widespread use of object-oriented programming.

Encapsulation and Inheritance

In a programming languages, encapsulation is used to refer to one of two related but distinct notions, and sometimes to the combination thereof:

A language mechanism for restricting access to some of the object's components.

A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.

Some programming language researchers and academics use the first meaning alone or in combination with the second as a distinguishing feature of object oriented programming, while other programming languages which provide lexical closures view encapsulation as a feature of the language orthogonal to object orientation.

The second definition is motivated by the fact that in many OOP languages hiding of components is not automatic or can be overridden; thus, information hiding is defined as a separate notion by those who prefer the second definition.Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. A benefit of encapsulation is that it can reduce system complexity, and thus increases robustness, by allowing the developer to limit the interdependencies between software components.Almost always, there is a way to override such protection – usually via reflection API (Ruby, Java, C#, etc.), sometimes by mechanism like name mangling (Python), or special keyword usage like friend in C++.

Below is an example in C# that shows how access to a data field can be protected through the use of a private keyword:

```
namespace Encapsulation

{

    class Program
     {
        public class Account
        {
```

```
        private decimal accountBalance = 500.00m;

        public decimal CheckBalance()
        {
            return accountBalance;
        }
    }

    static void Main()
    {
        Account myAccount = new Account();
        decimal myBalance = myAccount.CheckBalance();

        // This Main method can check the balance via the public
        // "CheckBalance" method provided by the "Account" class
        // but it cannot manipulate the value of "accountBalance"
    }
  }
}
```

Initialization and finalization

In object-oriented programming, a constructor (sometimes shortened to ctor) in a class is a special type of subroutine called at the creation of an object. It prepares the new object for use, often accepting parameters which the constructor uses to set any member variables required when the object is first created. It is called a constructor because it constructs the values of data members of the class.

Types of constructors

Parameterized constructors

Constructors that can take arguments are termed as parameterized constructors. The number of arguments can be greater or equal to one(1). For example

```
class example
{
    int p, q;
  public:
    example(int a, int b);                    //parameterized constructor
};
example :: example(int a, int b)
{
    p = a;
    q = b;
}
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.The method of calling the constructor implicitly is also called the shorthand method

```
example e = example(0, 50);                  //explicit call

example e(0, 50);                          //implicit call
```

Default constructors

If the programmer does not supply a constructor for an instantiable class, a typical compiler will provide a default constructor. The behavior of the default constructor is language dependent. It may initialize data members to zero or other same values, or it may do nothing at all.

Copy constructors

Copy constructors define the actions performed by the compiler when copying class objects. A copy constructor has one formal parameter that is the type of the class (the parameter may be a reference to an object).It is used to create a copy of an existing object of the same class. Even though both classes are the same, it counts as a conversion constructor.A constructor is a special type of method.

Dynamic method binding

The property of object-oriented programming languages where the code executed to perform a given operation is determined at run time from the class of the operand(s) (the receiver of the message). There may be several different classes of objects which can receive a given message. An expression may denote an object which may have more than one possible class and that class can only be determined at run time. New classes may be created that can receive a particular message, without changing (or recompiling) the code which sends the message. An class may be created that can receive any set of existing messages.

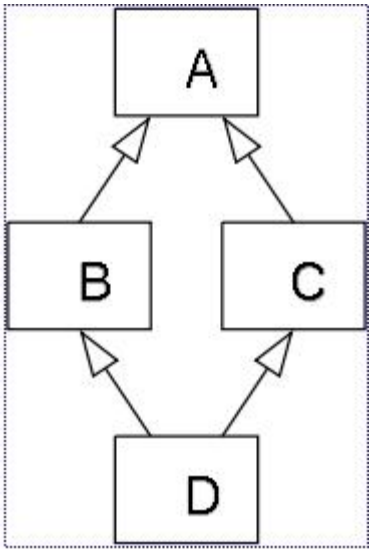
Multiple inheritances

Multiple inheritance is a feature of some object-oriented computer programming languages in which a class can inherit characteristics and features from more than one superclass. It is distinct to single inheritance, where a class may only inherit from one particular superclass.

The "diamond problem" (sometimes referred to as the "deadly diamond of death"[3]) is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If D calls a method defined in A (and does not override the method), and B and C have overridden that method differently, then from which class does it inherit: B, or C?

For example, in the context of GUI software development, a class Button may inherit from both classes Rectangle (for appearance) and Clickable (for functionality/input handling), and classes Rectangle and Clickable both inherit from the Object class. Now if the equals method is called for a Button object and there is no such method in the Button class but there is an overridden equals method in both Rectangle and Clickable, which method should be eventually called?

It is called the "diamond problem" because of the shape of the class inheritance diagram in this situation. In this article, class A is at the top, both B and C separately beneath it, and D joins the two together at the bottom to form a diamond shape.


A
B
C
D