# What is Programming Language

## Table of Contents



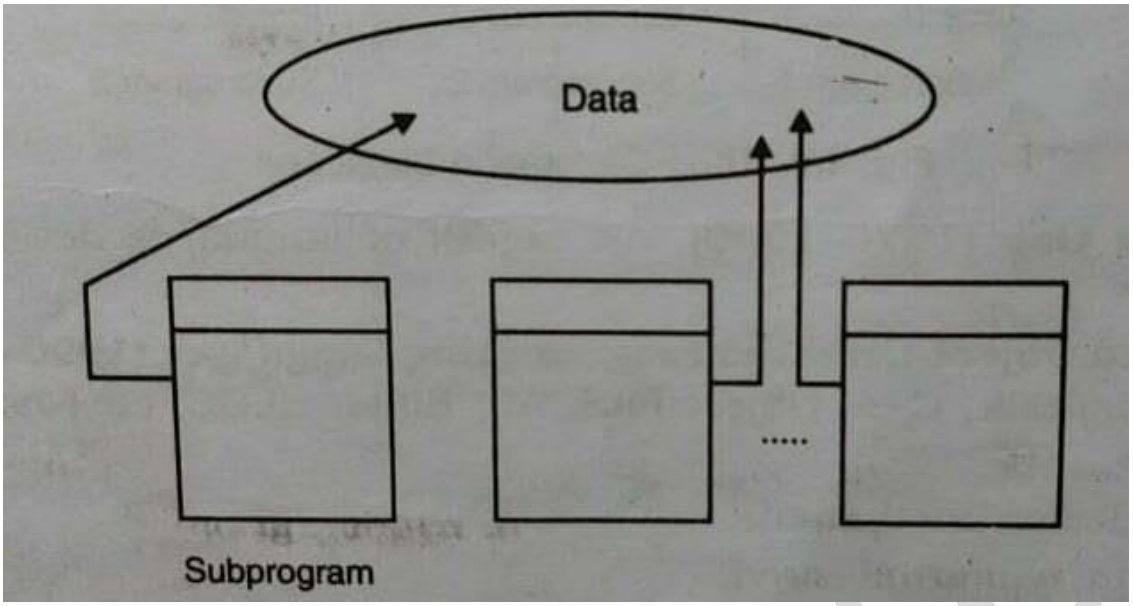
## 1.1 Introduction

Programming languages may be thought to be belonging to different generations, depending on the way a program is structured and the way data and program are connected.

### First-Generation Languages (1954-1958):

These first-generation languages (Fortan 1, ALGOL 58) have the following features:

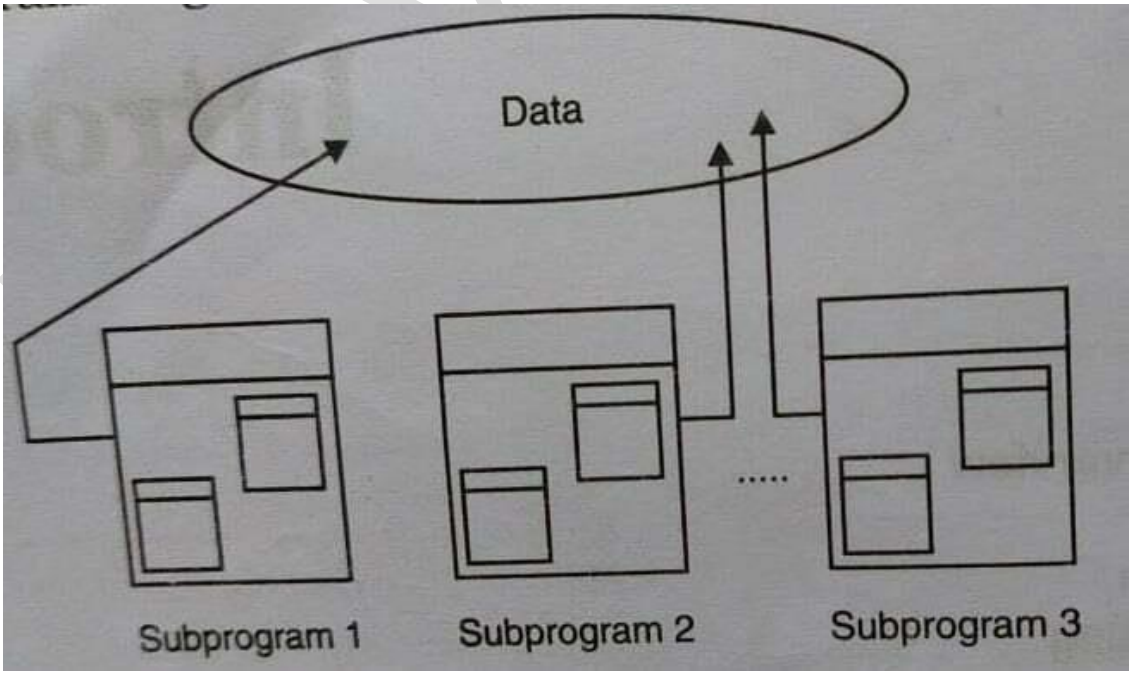
1.
    1. Subprograms were seen as more labor-saving devices.
    2. Data were globally defined.




## Second-Generation Languages (1959-1961):

To this generation belong such languages as Fortran 11, Algol 60, COBOL, and LISP. They have the following features

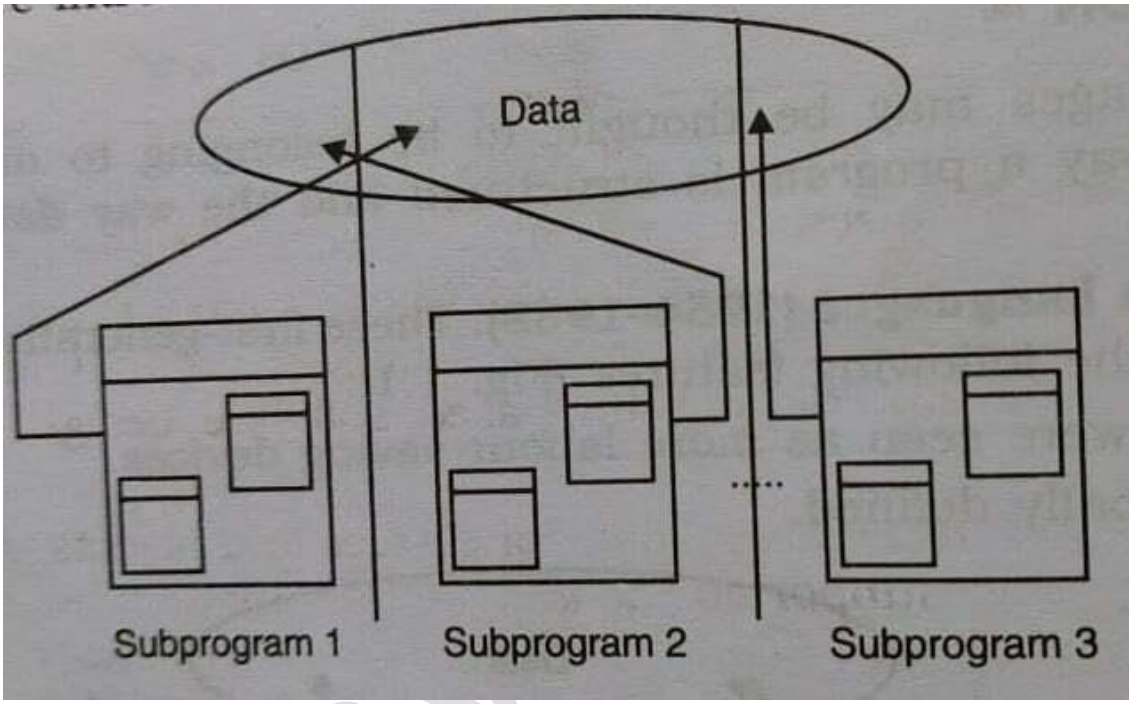
- 
  - Nesting of subprograms was allowed.
  - Various methods were used for passing parameters from are subprogram to another.
  - Structured programming constructs were used.



## Third-Generation Languages (1962-1970):

The languages belonging to this generation are PL/1, ALGOL 68, PASCAL, and Simula. The features of these languages are as under:

- 
  - Programming-in-the-large
  - Separately compiled modules
  - Data types were introduced

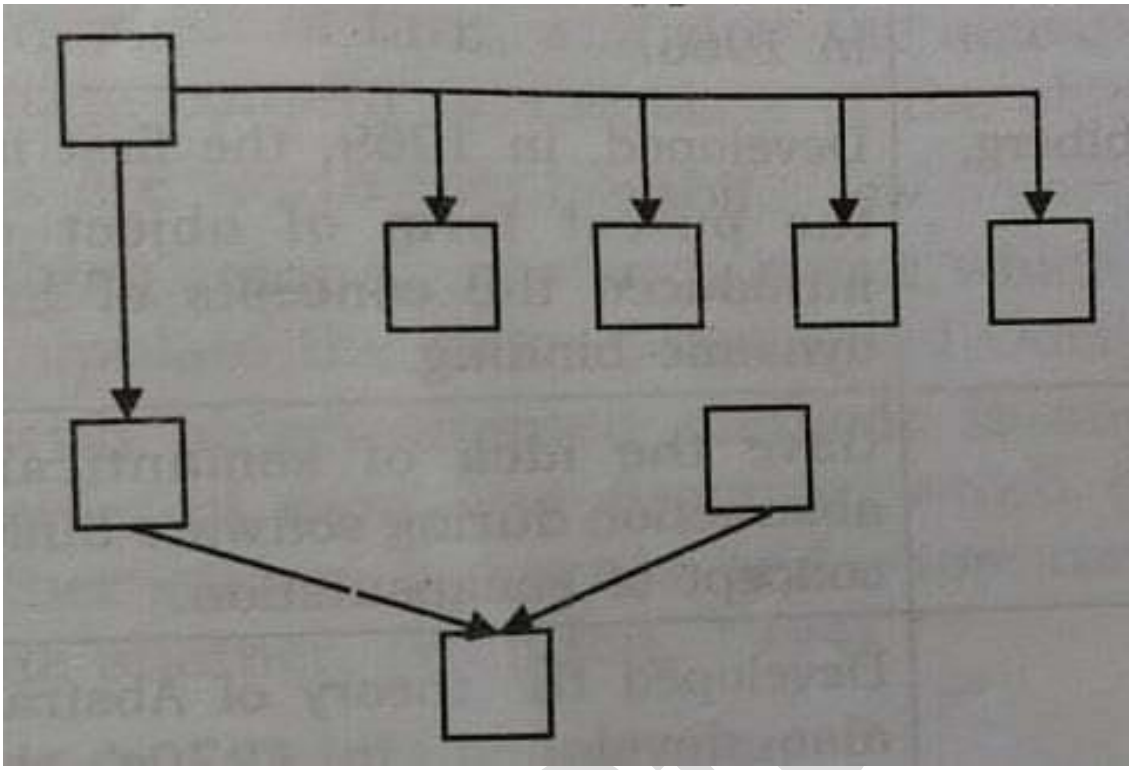



The Generation Gap (1970-1990). An excess of languages developed during the seventies.

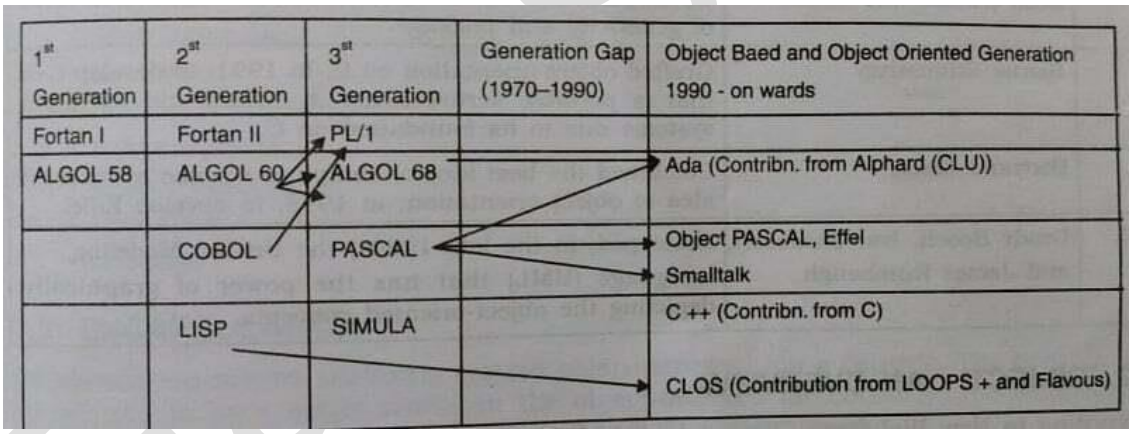
## Object-Based and Object-Oriented Programming Languages (1990-Onwards):

These languages (Ada, Smalltalk, C++, Object PASCAL, Eiffel, CLOS, etc.) have the following features

1. Data-driven design methods.
2. The theory of data typing emerged.
3. Little or no global data.
4. The physical structure of an application appears like a graph, rather than a tree.



**Evolution of generation of Languages**




One way to distinguish a procedure-oriented language from an object-oriented language is that the former is organized around procedures and functions (verbs) whereas the latter is organized around pieces of data (nouns). Thus, in a procedure-oriented program-based design, a module represents a major function, such as 'Read a Master Record', whereas in an object-oriented program-based software design, 'Master Record' is a module.

Simula-80 had the fundamental ideas of classes and objects. Alphard, CLU, Euclid, Gypsy, Mesa, and Modula supported the idea of data abstraction. The use of object-oriented concepts led to the development of C to C++; Pascal to Object Pascal, Eiffel, and Ada; LISP to Flavors, LOOPS, and Common LISP Object System (CLOS)

The concepts of object orientation came from many computer scientists in different areas of computer science. We give below, almost chronologically, a list of prominent scientists whose contributions were significant.

| Sr. No. | Computer Scientist | Contribution |
|---|---|---|
| **1** | Larry Constantine | Gave the idea of coupling and cohesion in the 1960s that provided the principles of the modular design of programs. |
| **2** | D. J. Dahl and K. Nygaard | Introduced the concept of 'class' in the language Simula in 1966. |
| **3** | Allan Kay, Adele Goblberg, and others | Developed, in 1969, the first incarnation of Smalltalk and others the purest form of object-orientation where they introduced the concepts of inheritance, message, and dynamic binding. |
| **4** | Edsger Dijkstra | Gave the idea of semantically separated layers of abstraction during software building, which is the central concept of encapsulation. |
| **5** | Barbara Liskov | Developed the theory of Abstract Data Type (ADT) and also developed, in the 1970s, the CLU Language that supported the notion of hidden internal data representation. |
| **6** | David Parnas | Forwarded the principle of information hiding in 1972. |
| **7** | Jean Ichbiah and others | Developed ADA that had, for the first time, the features of genericity and package. |
| **8** | Bjarne Stroustrup | Grafted object orientation on C, in 1991, to develop C++ that is portable across many machines and operating systems due to its foundation on C. |
| **9** | Bertrand Meyer | Combined the best idea of computer science to the best idea of object orientation, in 1995, to develop Eiffel. |
| **10** | Grady Booch, Ivar Jacobson, and James Rumbaugh | Developed, in the late 1990s, the Unified Modelling Language (UML) has the power of graphically depicting object-oriented concepts. |

## 1.2. 'Object': An Overview

According to New Webster's Dictionary (1981), an object is :

- some visible or tangible thing
- hat toward which the mind is directed in any of its states or activities;
- that to which efforts are directed.

Thus an object refers to a thing, such as a chair, a customer, a university, a painting, a plan, or a mathematical model. The first four of these examples are real-world objects, while the last two are conceptual or abstract objects. Software engineers build abstract objects that represent real-world objects that are of interest to a user.

In the context of object-oriented methodologies, the second dictionary definition is more appropriate: An object is anything, real or abstract, which is characterized by the *activities* defined on that object that can bring about change in the state.

In a later section, we shall expand the idea of the state and the activities. Suffice it is to state at this point that the state of an object indicates the information the object stores within itself at any point in time and that the activities are the operations and can change the information content or the state of the object.

Two other definitions are worth mentioning:

- An object is anything, real or abstract, about which we store data and those methods that manipulate the data. (Martin and Odell, 1992).
- A system built with object-oriented methods is one whose components are encapsulated chunks of data and function, which can inherit attributes and behavior from other such components, and whose components communicate via messages with one another. (Yourdon, 1994).

### Central Concepts Underlying Object Orientation

Various authors have suggested various concepts that, they think, are central to object orientation. We give below some of the oft-repeated concepts:

- Encapsulation
- Classes
- State retention
- Inheritance
- Information hiding
- Polymorphism
- Object identity
- Genericity
- Message

## 1.3. Encapsulation

Encapsulation means enclosing related components within a capsule. The capsule can be referred to by a single name. In the object-oriented methodology, the components within this capsule are (1) the attributes and (2) the operations. Attributes store information about the object. The operations can change the values of the attributes and help access them.

## 1.4. State Retention

The idea of encapsulation is not unique to object orientation. Subroutines in early high-level languages had already used the idea of encapsulation. Modules in structured design also represent encapsulation. There is however a difference between encapsulation represented in modules and that in objects. After a module completes its tasks, the module returns back to its original state. In contrast, after an operation is performed on an object, the object does not return to its original state; instead, it continues to retain its final state till it is changed when another operation is performed on it.

## 1.5. Information Hiding

One result of encapsulation is that details of what takes place when an operation is performed on an object are suppressed from public view. Only the operations that can be performed on an object are visible to an outsider. It has two major benefits:

1. It localizes design decisions. Private design decisions (within an object) can be made and changed with minimal impact upon the system as a whole.
2. It decouples the content of information from its form.

Once again the idea of information hiding is not new. This idea was forwarded by Parnas (1972) and was used in the modular design of programs in structured design.

## 1.6. Object Identity

Every object is unique and is identified by an object reference or an object handle. A programmer can refer to the object with the help of such a reference (or handle) and can manipulate it. Thus a program statement

***Customer cust_rec 1= new Customer ()***;

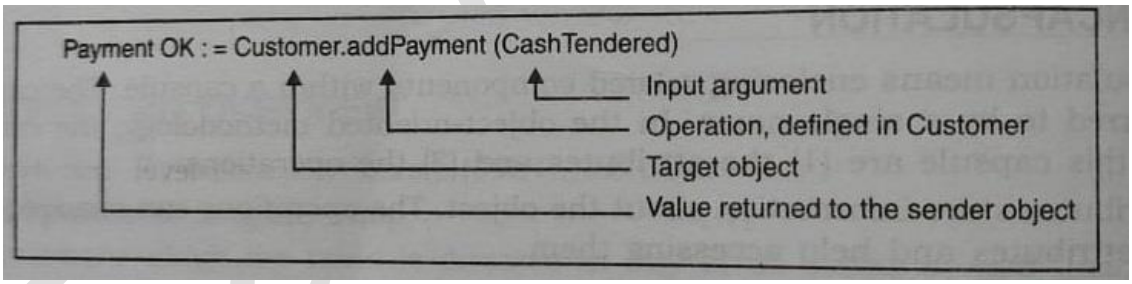
defines a variable cust-rec 1 and causes this variable to hold the handle of the object Customer, created newly. This object belongs to the class customer. The assignment operator (-) directs the class Customer to create an instance of its own.

## 1.7. Messages

An object obj1 requests another object obj2, via a message, to carry out an activity using one of the operations of obj2. Thus obj1 should

1. Store the handle of obj2 in one of its variables.
2. Know the operation of obj2 that it wishes to execute.
3. Pass any supplementary information, in the form of arguments, that may be required by obj2 to carry out the operation.

Further, obj2 may pass back the result of the operation to obj1.




The input arguments are generally parameter values defined in (or variable at) obj 1. But they can also be other objects as well. In fact, in Smalltalk, there is no need for any data. Objects point to other objects (via variables) and communicate with one another by passing back and forth handles of other objects.

**Messages can be of three types:**

1. Informative (past-oriented, update, forward, or push)
2. Interrogative (present-oriented, real, backward, or pull)
3. Imperative (further-oriented, force or action).

An informative message provides the target object information on what has taken place elsewhere in order to update itself:

***employee.updateAddress (address:Address)***

An interrogative message requests the target object for some current information about itself:

***inventory.getStatus***

An imperative message asks the object to take some action in the immediate future on itself, another object, or even on the environment around the system.

***payment.computeAmount (quantity, price)***

## 1.8. Classes

A class is a stencil from which objects are created (instantiated); that is, instances of a class are objects. Thus customer 1, customer 2, and so on, objects of the class customer, and product 1, product 2, are objects of the class product.

The UML definition of a class is "a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics". It does not include concrete software implementation such as a Java class; thus it includes all specifications that precede implementation. In the UML, an implemented software class is called an implementation class.

Oftentimes a term type is used to describe a set of objects with the same attributes and objects. Its difference from a class is that it may not include any methods. Recall that a method is the implementation of an operation, specifying the operation's algorithm or procedure.

Although objects of a class are structurally identical, each object:

1. has a separate handle or reference and
2. can be in different states.

Normally, operations and attributes are defined at the object level, but they can be defined at the level of a class. Thus, creating a new customer is a class-level operation:

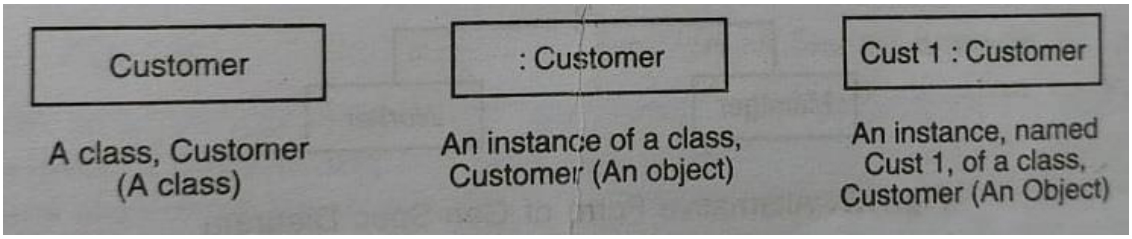
***new Customer();***

Customer() is a class operation that creates a new customer (constructor). Similarly, no0fCustomers Created that keeps a count of the number of customer objects created by the class customer is a class-level attribute:

***noOfCustomersCreated: Integer***

*no0fCustomers* Created is a class attribute that is incremented by 1 each time the operation New is executed.

The UML notation of a class, an instance of a class, and an instance of a class with a specific name are as under:




## 1.9. Inheritance

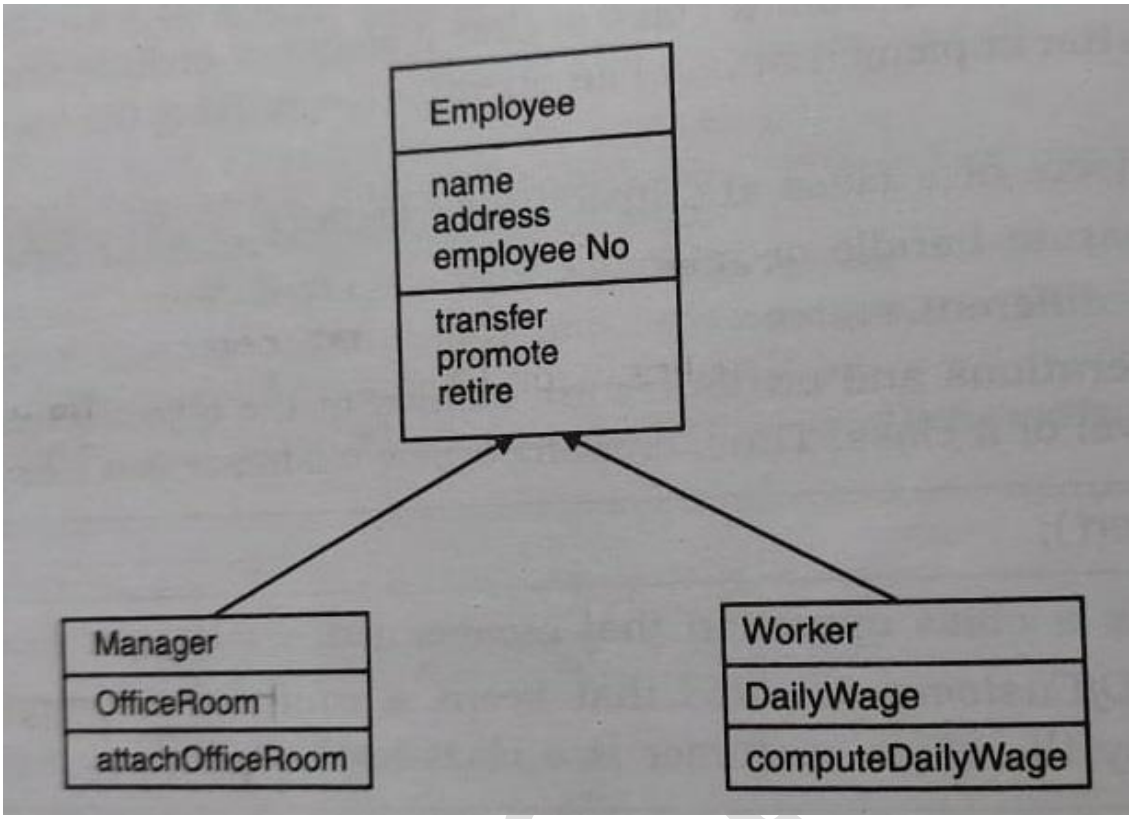
Inheritance (by D from C) is a facility by which a subtype D implicitly defines upon it all the attributes and operations of a supertype C as if those attributes and operations had been defined upon D itself.

Note that we have used the terms subtypes and supertypes instead of the terms subclasses and superclasses (although the latter two terms are popularly used in this context) because we talk of only operations and attributes, and not meth.

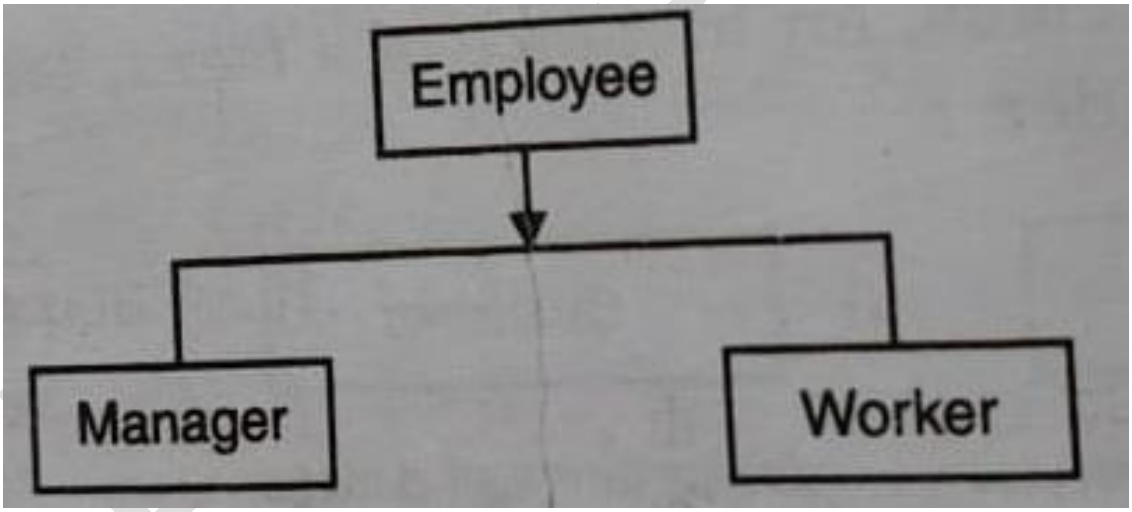
The classes Manager and Worker are both Employees. So we define attributes such as Name, Address, and EmployeeNo, and define operations such as transfer, promote, and retire in the supertype Employee. These attributes and operations are valid for and can be used by, the subtypes, Manager, and Worker, without separately defining them for these subtypes. In addition, these subtypes can define attributes and operations that are local to them. For example, an attribute OfficeRoom and operation attachOfficeRoom can be defined on the Manager, and an attribute DailyWage and an operation. computeDailyWage can be defined on Worker.

Inheritance is best depicted in the form of a Gen-Spec (Generalization-Specialization) diagram. The example of a Manager and Worker inheriting from an employee is depicted below in the form of a Gen-Spec diagram.

Here, Employee is a generalized class, and Manager and Worker are specialized classes.



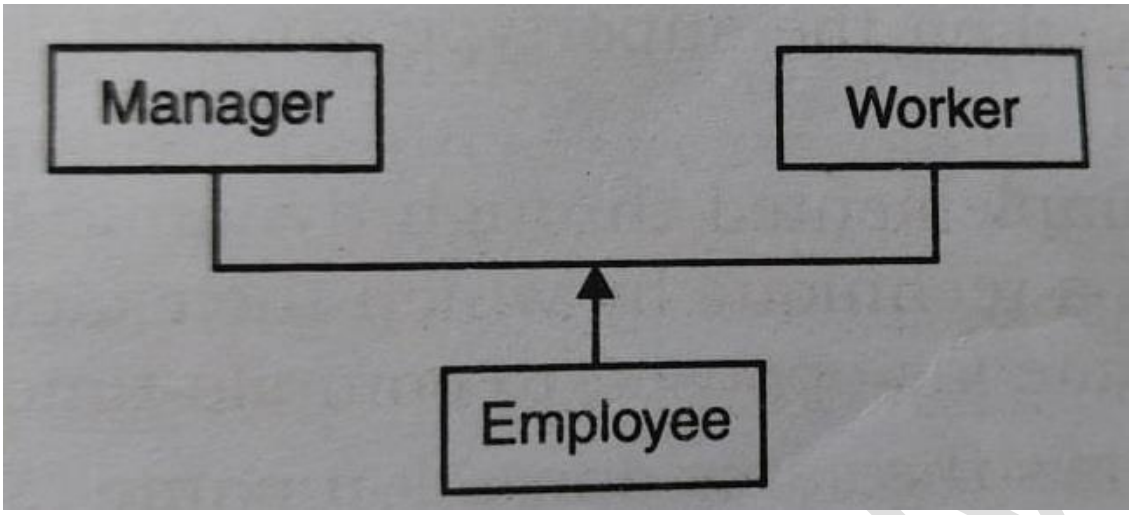

An alternative UML notation is given below




To define a correct subtype, two rules are to be satisfied:

1. The 100% Rule. The subtype conforms to 100% of the super-type's attributes and operations.
2. The Is-a Rule. The subtype is a member of the super-type.

The Gen-spec diagram is often called an 'Is-a' diagram. Often a subtype can inherit attributes and operations from two super-types. Thus a Manager can be both an Employee and a Shareholder of a company. This is a case of multiple inheritances.



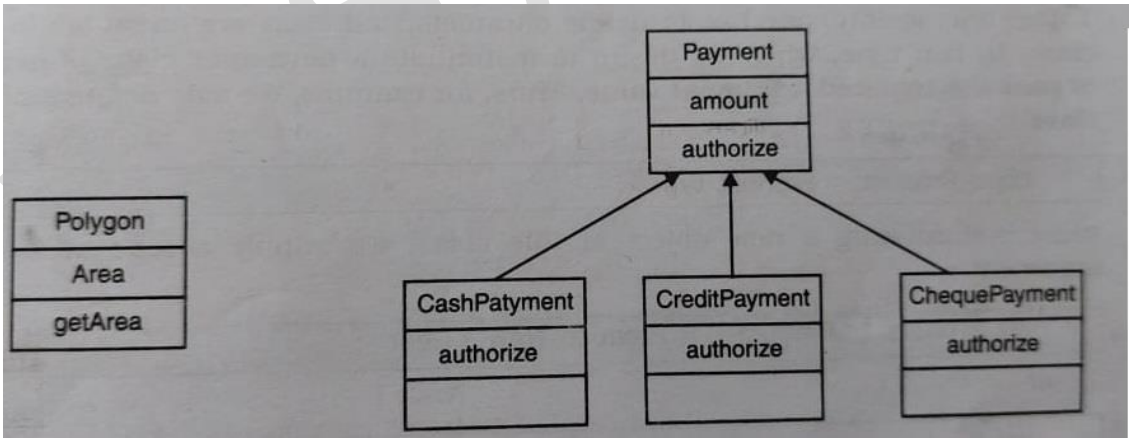

While languages such as C++ and Eiffel support these features, Java and Smalltalk do not. Multiple inheritances lead to problems of :

1. Name-clash
2. Incomprehensibility of structures

More about their problems will be cited later in the text.

## 1.10. Polymorphism

Polymorphism is a Greek word, with poly meaning 'many' and morph meaning 'form'. Polymorphism allows the same name to be given to services in different objects when the services are similarly related. Usually, different object types are related in a hierarchy with a common supertype, but this is not necessary (especially in dynamic binding languages, such as Smalltalk, or languages that support interface, such as Java).




In the first example, getArea is an operation in the supertype Hexagon that specifies a general method of calculating the area of a Polygon. The subtype Hexagon inherits this operation, and therefore the method of calculating its area. But if the polygon happens to be a Triangle or a Rectangle, the same operation getArea would mean calculating the area by simpler methods such as ½ (product of the base and the height), or product of two adjacent sides, respectively.

In the second example, Payment types are different cash, credit, or cheque. The same operation authorization is implemented differently in different Payment types. In CashPayment, authorize looks for counterfeit paper currency; in CreditPayment, it checks for creditworthiness; and in ChequePayment, it examines the validity of the cheque.

In these two examples, the concept of overriding has been used. The operations getArea and authorize defined on the supertype are overridden in the subtypes, where different methods are used.

Polymorphism is often implemented through dynamic binding. Also called run-time binding or late binding, it is a technique by which the exact piece of code to be executed is determined only at run-time (as opposed to compile-time), when the message is sent.

While polymorphism allows the same operation name to be defined differently across different classes, a concept called overloading allows the same operation name to be defined differently several times within the same class. Such overloaded operations are distinguished by the signature of the message, Le., by the number and/or class of the arguments. For example, two operations, one without an argument and the other with an argument, may invoke different pieces of code:

***giveDiscount***

***giveDiscount (percentage)***

The first operation invokes a general discounting scheme allowing a standard discount percentage, while the second operation allows a percentage discount that is specified in the argument of the operation.

## 1.11. Genericity

Genericity allows defining a class such that one or more of the classes that it uses internally is supplied only at run time, at the time an object of this class is instantiated. Such a class is known as a parameterized class. In C++ it is known as a template class. To use this facility, one has to define parameterized class argument while defining the class. In run time, when we desire to instantiate a particular class of items, we have to pass the required argument value. Thus, for example, we may define a parameterized class:

***class Product <Product type>;***

while instantiating a new object of this class, we supply a real class name as an argument:

***var Product 1: Product Product. New <Gear>;***

or

***var Product 2: Product Product. New <Pump>;***