UNIT 2 OBJECT-ORIENTED METHODOLOGY- 2

Structure **Page Nos.**

2.0 INTRODUCTION

Object-oriented is a philosophy for developing software. In this approach we try to solve the real-world problems using real-world ways, and carries the analogy down to the structure of the program. In simple words we can say the Object-Oriented Methodology is a method of programming in which independent *blocks of codes* or *objects* are built to interact with each other similar to our real-world objects. It is essential to understand the underlying concepts related to an object before you start writing objects and their methods. You need to understand **How objects and classes are related?** and **How objects communicate by using messages?**

OOP
⇩
Type-extensibility
+
Polymorphism

In this unit the concepts **behind** the object-oriented methodologies will be described, which form the **core** of Object-oriented languages like Java and C++.

We will start with a basic concept of *class and object*, then we will learn the concept of **abstraction** and **encapsulation** in relation to the object-oriented methodology. We will also discuss the very important concept of **inheritance**, which helps in *fast* and *efficient* development of programs. Finally we will discuss the concept of **polymorphism**.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- define class and object;
- explain the concept of abstraction and encapsulation;
- explain the concept of inheritance, and
- explain method overriding and polymorphism.

2.2 CLASSES AND OBJECTS

Before we move further, in the very first instance let's define what is object-oriented programming.

Definition: OOP is an approach that provides a way of modularizing programs by creating partitioned memory blocks for both data and functions that can be used as templates for creating copies of such modules on demand.

This means that an object is considered to be a block of computer memory that stores **data** and a set of **operations** that can access the stored data. Since memory blocks are independent, the objects can be used in a variety of different programs without modifications. To understand this definition, it is important to understand the notion of objects and classes.

Now let us discuss the notion of objects and classes.

Fig. 1a: Car a real World Object

Objects

Objects are the key to understand object-oriented technology. You can look around and can see many examples of real-world objects like your car, your dog or cat, your table, your television set, etc. as shown in *Figure 1a and 1b*.

Fig. 1b: A real World Object

These real-world objects share two characteristics: They all have *state* and *behavior*. For example, **dogs** have state (name, color, breed, hungry) and behavior (barking, fetching, and wagging tail). **Cars** have state (current gear, current speed, front wheels, number of gears) and behavior (braking, accelerating, slowing down, changing gears).

Because Software objects are modeled after real-world objects, so in that they too have state and behavior. A software object maintains its state in one or more variables. A variable is an item of data named by an identifier. A software object reflects its behavior with the help of method implementation. A method is an implementation way of a function associated with an object.

Definition: *An object is a software bundle of variables and related methods*. You can represent real-world objects by using software objects. For example, you might have observed that real world dogs as software objects in an animation program or a real-world airplane as software object in the simulator program that controls an electronic airplane. You can also use software objects to model abstract concepts.

Variable is a symbol that can hold different values at different times.

For example, an *event* is a common object used in GUI window systems to represent the action of a user pressing a mouse button or a key on the keyboard. The illustration given in *Figure 2,* is a common visual representation of a software object:

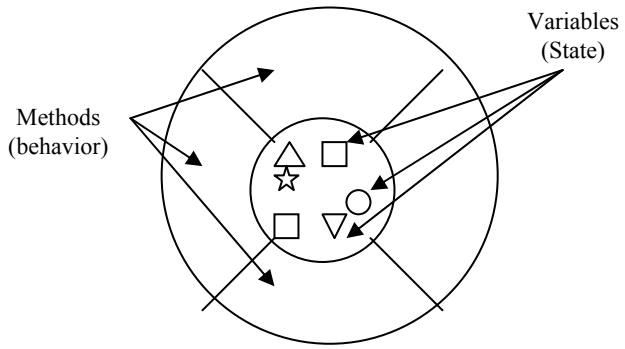
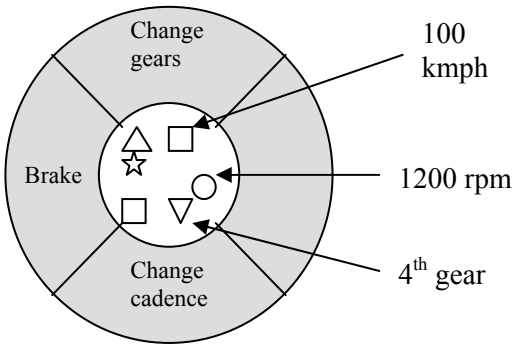


Figure 2: A Software Object

Everything that the software object *knows* (state) and *can do* (behavior) is expressed by the variables and the methods within that object as shown in *Figure 2*. For example, a software object that modeled your real-world **car** would have variables that indicated the car's *current state*: as its speed is 100 kmph, and its current gear is the 5th gear. These variables used to represent the object's state are formally known as **instance variables**, because they contain the state for a particular car object. In object-oriented terminology, a particular object is called an *instance* of the class to which it belongs. In *Figure 3* it *is* shown how a car object is modeled as a software object.



Instance variable: variables of a class, which may have a different value for each object of that class.

Figure 3: A Car Object

In addition to its variables, the software **car** would also have methods to brake, change the cadence, and change gears. These methods are formally known as instance methods because they impact or change the state of a particular car instance.

Class

In the real world, you often have many objects of the same kind. *For example*, *your car* is just one of many cars in the world. In object-oriented terminology, we say that your car object is an instance of the class known as car. Cars have some state (current gear, current cadence, front and rear wheels) and behavior (change gears, brake) in common. However, each car's state is independent of the other and can be different from other vehicles.

When building cars, manufacturers take advantage of the fact that cars share characteristics, building many cars from the same blueprint. It would be very inefficient to produce a new blueprint for every individual car manufactured. The blueprint is a template to create individual car objects.

A software blueprint for objects is called a class.

In object-oriented software, it is also possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips, and so on. Like the car manufacturers, you can take advantage of the fact that objects of the same kind are similar and you can create a blueprint for those objects.

Definition: *A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.*

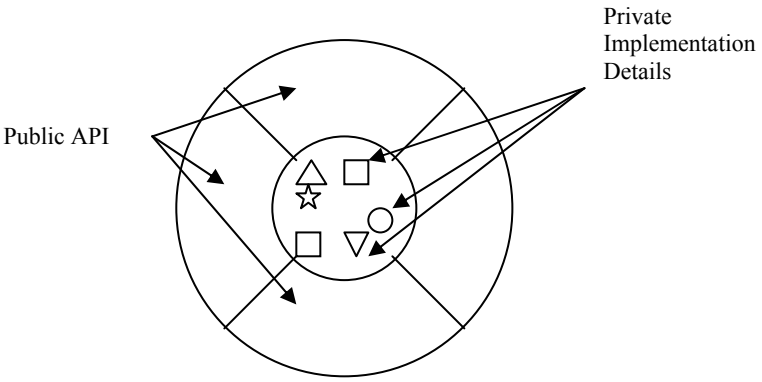


Figure 4: A Class Structure

The class for our car example would declare the instance variables necessary to contain the current gear, the current speed, and so on, for each car object. The class

would also declare and provide implementations for the instance methods that allow the driver to change gears, brake, and show the speed, as shown in *Figure 5.*

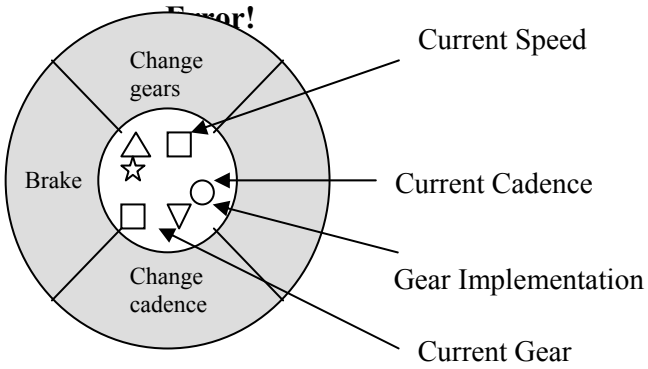


Figure 5: Structure of a "car" Class

After you've created the car class, you can create any number of car objects from the class. For example, we have created two different objects Your Car and My Car (as shown in *Figure 6)* from class car. When you create an instance of a class, the system allocates **enough memory** for the object and all its instance variables. Each instance gets its own copy of all instance variables defined in the class as shown in *Figure 6.*

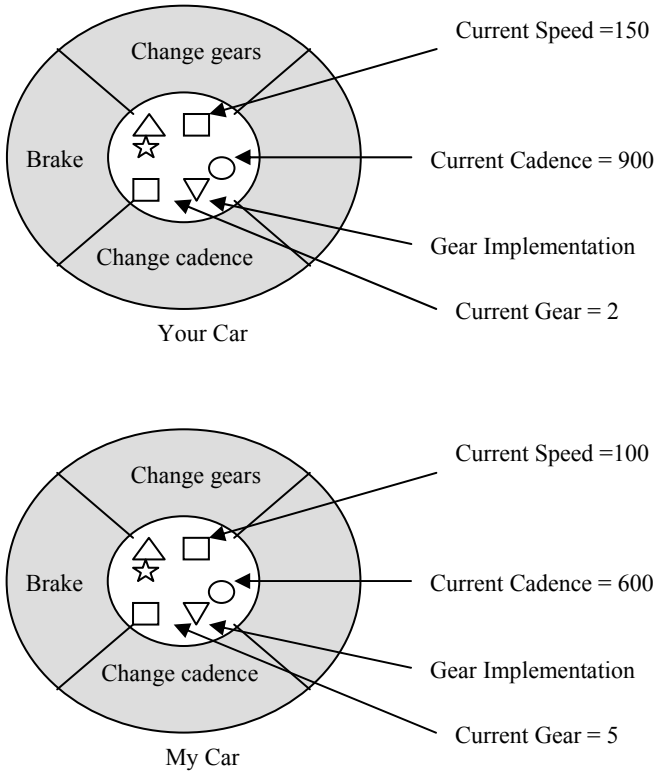


Figure 6: Instance variables of class

In addition to instance variables, **classes can define class variables**. A class variable contains information that is shared by all instances of the class. **For example**, suppose that all cars had the same number of gears (18) as shown in *Figure 7a*. In this case, defining an instance variable to hold the number of gears is inefficient; each instance would have its own copy of the variable, but the value would be the same for every instance. In such situations, you can define a **class variable** that contains the number of gears. All instances share this variable as shown in *Figure 7b*. If one object changes the variable, it changes for all other objects of that class. A class can

also declare class methods. You can invoke a class method directly from the class, whereas you must invoke instance methods in a particular instance.

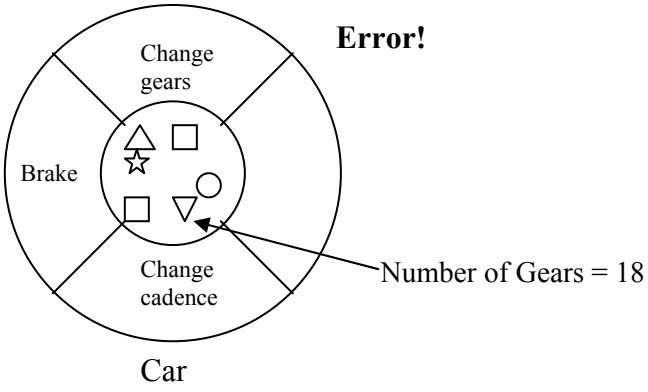


Figure. 7a: Class

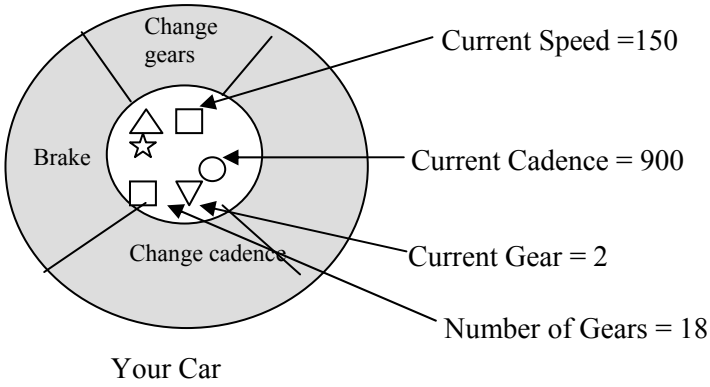


Figure 7b: Instance of a Class

Objects vs. Classes

You probably noticed that the illustrations of objects and classes look very similar. And indeed, the difference between classes and objects is often the source of some confusion. In the real world, it's obvious that classes are not themselves the objects they describe: A blueprint of a car is not a car. However, it's a little more difficult to differentiate classes and objects in software. This is partially because software objects are merely electronic models of real-world objects or abstract concepts in the first place. But it's also because the term "**object"** is sometimes used to refer to both classes and instances.

A single object alone is generally not very useful. Instead, an object usually appears as a component of a larger program or application that contains many other objects. Through the interaction of these objects, programmers achieve higher-order functionality and more complex behavior. Your car standing in the garage is just a product of steel and rubber. By itself the car is incapable of any activity. The car is useful only when another object (may be you) interacts with it (clutch, accelerator).

Software objects interact and communicate with each other by sending **messages** to each other. As shown in *Figure 8* when object **A** wants object **B** to perform one of **B's** methods, object **A** sends a message to object **B** for the same.

Error!

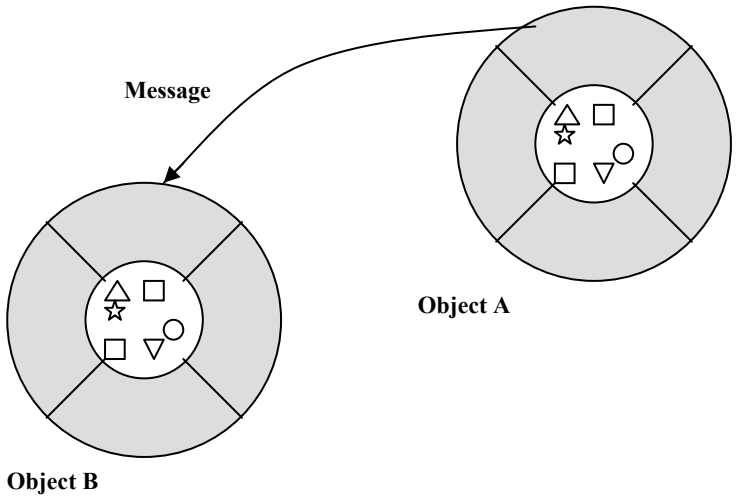


Figure 8: Message passing between objects

Sometimes, the receiving object needs more information so that it knows exactly what to do; for example, when you want to change gears on your car, you have to indicate which gear you want. This information is passed along with the message as *parameters*. The *Figure 9* shows the **three components** that comprise a message:

1. The object to which the message is addressed (Your Car)
2. The name of the method to perform (change Gears)
3. Any parameters needed by the method (lower Gear)

Change Gears (lowerGear)

Error!

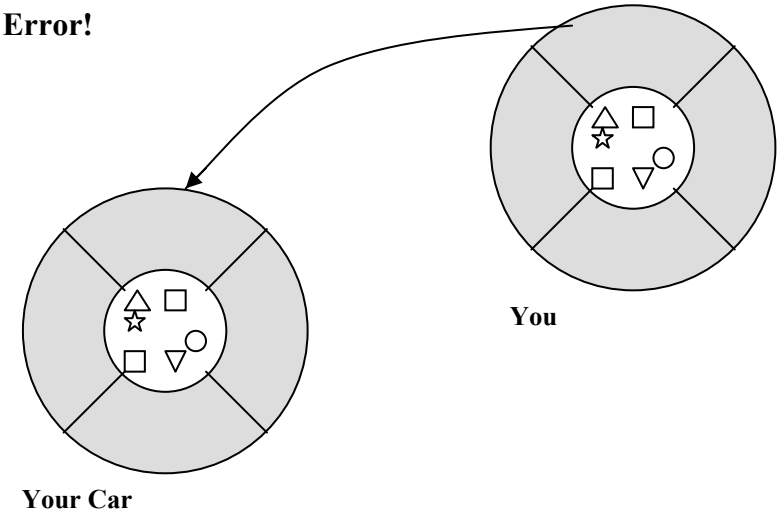


Figure 9: Components of message

These three components contain enough information for the receiving object to perform the desired method. No other information or context is required. Messages provide **two important benefits**.

- An object's behavior is expressed through its methods, so (aside from direct variable access) message passing supports all possible interactions between objects.
- Objects don't need to be in the same process or even on the same machine to send and receive messages back and forth to each other.

☞ Check Your Progress 1

1) What is instance variable? Two different objects of same class can have same value for an instance variable or not?

……

……

……

2) How object and class are associated with each other?

……

……

……

3) Why an object can be used in different programs without modification?

……

……

2.3 ABSTRACTION AND ENCAPSULATION

"*Encapsulation is used as a generic term for techniques, which realize data abstraction. Encapsulation therefore implies the provision of mechanisms to support both modularity and information hiding. There is therefore a one-to-one correspondence in this case between the technique of encapsulation and the principle of data abstraction" -- [Blair et al, 1991]*

The object diagrams show that the object's variables make up the center, or nucleus, of the object. Methods surround and hide the object's nucleus from other objects in the program. *Packaging* an object's variables within the protective custody of its methods is called **encapsulation**. This conceptual picture of an object-a nucleus of variables packaged within a protective membrane of methods-is an ideal representation of an object and is the ideal that designers of object-oriented systems strive for.

However, it's not the whole story. Often, for practical reasons, an object may wish to expose some of its variables to other objects or hide some of its methods from others. In the Java programming language, an object can specify one of four access levels for each of its variables and methods. This feature of Java we will study in Unit 2 of Block 2 of this course. The access level determines which other objects and classes can access the variables or methods of object. Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

- **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your car to someone else, and it will still work.
- **Information hiding:** An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. You don't need to understand the gear mechanism on your car to use it.

Therefore, a class is a way to bind the data and its associated methods together. It allows the data (and functions) to be hidden, if necessary, from external use.

Abstraction

"*A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information*.'' *-- [IEEE, 1983]*

Abstraction refers to the act of representing essential features without including the background details. All programming languages provide abstractions. Assembly language is a small abstraction of the underlying machine. Procedural languages that followed (such as FORTRAN, BASIC, and C) were abstractions of assembly language. These languages are big improvements over assembly language, but their primary abstraction still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve. The programmer must establish the association between the machine model (in the "solution space", which is the place *where you're modeling that problem* i.e. computer) and the *model of the problem* that is actually being solved (in the "problem space", which is a place where the problem exists). A lot of effort is required to perform this mapping and it produces programs that are difficult to write and expensive to maintain.

The object-oriented approach provides tools for the programmer to represent elements in the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. Here we refer to the elements in the problem space and their representations in the solution space as "objects". Object-oriented programming (OOP) allows you to describe the problem in terms of the problem, rather than in terms of the solution or computer where the solution will be executed.

Abstract Data Types: A set of data values and associated operations that are defined independent of any particular implementation.

In object-oriented approach, classes use the concept of **data abstraction**. With data abstraction data structures can be used without having to be concerned about the exact details of implementation. As in case of **built-in data types** like integer, floating point etc. the programmer only knows about the various operations which can be performed using these data types, but how these operations are carried out by the hardware or software is hidden from the programmer.

Classes act as ***abstract data types***. Classes are defined as a set of abstract attributes and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created.

When defining a class, we are creating a new *abstract data type* that can be treated like any other built-in data type. Generally, a class specification has two parts:

i. **Class declaration**
ii. **Class method definition**

The class declaration describes the type and scope of its members. The class method definitions describe how the class functions are implemented. We will study about them in detail in the later units of this block.

2.4 INHERITANCE

Now, you are conversant with classes, the building blocks of OOP. Now let us deal with another important concept called **inheritance**. Inheritance is probably the most powerful feature of object-oriented programming.

Inheritance is an ability to derive new classes from existing classes. A derived class is known as subclass which inherits the instance variables and methods of the super

class or base class, and can add some new instance variables and methods. [Katrin Becker 2002].

Generally speaking, objects are defined in terms of classes. You know a lot about an object by knowing its class. If I tell you that the object is a bicycle, you can easily tell that it has two wheels, a handle bar, and pedals.

Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. For example, mountain bikes, racing bikes, and tandems are all kinds of bicycles as shown in *Figure 10.* In OOP, mountain bikes, racing bikes, and tandems are all subclass (i.e. derived class or child class) of the bicycle class. Similarly, the bicycle class is the superclass (i.e., base class or parent class) of mountain bikes, racing bikes, and tandems. This relationship you can see shown in the *Figure 10.*

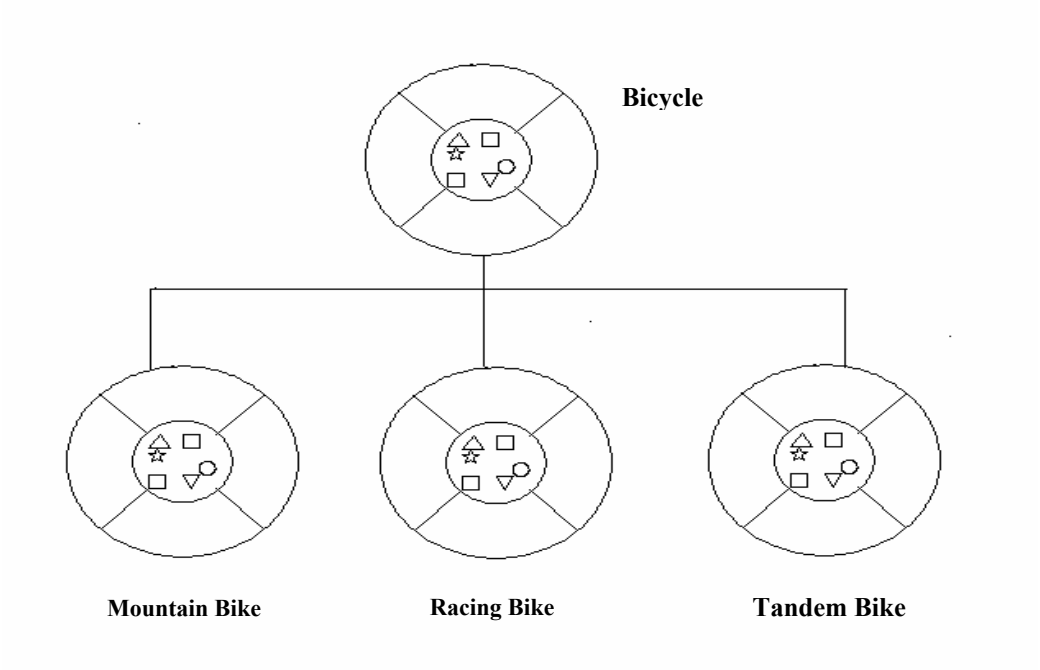


Figure 10: Superclass and Subclasses

Each subclass inherits state (in the form of variable declarations) from the superclass. Mountain bikes, racing bikes, and tandems share some states: cadence, speed, etc. Also, each subclass inherits methods from the superclass. Mountain bikes, racing bikes, and tandems share some behaviors: for example braking and changing pedaling speed.

However, subclasses are not limited to the state and behaviors provided to them by their superclass. Subclasses can add variables and methods to the ones they inherit from the superclass. Tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an extra set of gears with a lower gear ratio.

Subclasses can also override inherited methods and provide specialized implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the change gears method so that the rider could use those new gears.

You are not limited to just one layer of inheritance. The inheritance tree, or class hierarchy can be as deep as needed. Methods and variables are inherited down through the levels. In general, the farther down in the hierarchy a class appears, the more specialized its behavior. Inheritance may have different forms as shown in *Figure 11*:

- **Single Inheritance (Fig. 11a)**: In this form, a subclass can have only one super class.
- **Multiple Inheritance (Fig. 11b)**: This form of inheritance can have several superclasses.
- **Multilevel Inheritance (Fig. 11c)**: This form has subclasses derived from another subclass. The example can be grandfather, father and child.
- **Hierarchical Inheritance (Fig. 11d)**: This form has one superclass and many subclasses. More than one class inherits the traits of one class. For example: bank accounts**.**

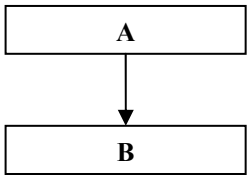


Figure 11a: Single Inheritance

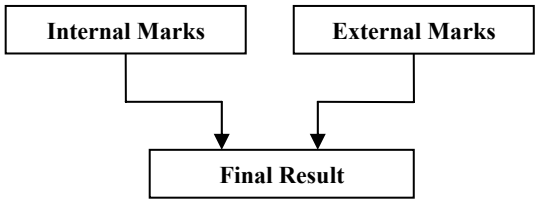


Figure 11b: Multiple-Inheritance

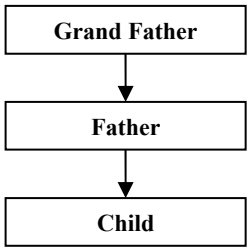


Figure 11c: Multilevel Inheritance

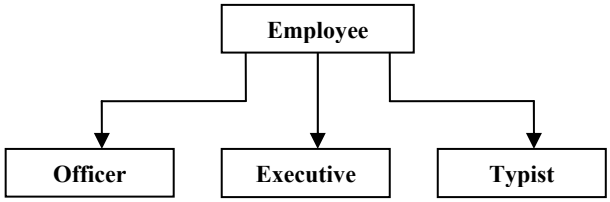


Figure 11d: Hierarchical Inheritance

Figure 11: Different forms of Inheritance

These forms of inheritance could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance. *For example*, in case of single inheritance, traits of class **A** are inherited by class **B**.

☞ Check Your Progress 2

1) What is data abstraction and why classes are known as abstract data types?

……

……

……

2) What are the two main benefits provided by encapsulation to software developers?

……

……

……

3) What is inheritance? Differentiate between multilevel and multiple inheritance.

……

……

……

……

Benefits of Inheritance

- Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times. Once a superclass is written and debugged, it need not be touched again but at the same time can be adapted to work in different situations. Reusing existing code saves time and money and increases a program's reliability.
- Programmers can implement superclasses called abstract classes that define "*generic*" behaviors. A class for which objects doesn't exist. Example: Further class has no object, but its derived classes-chair, table-that have objects. The abstract superclass defines and may partially implement the behavior, but much of the abstract class is undefined and unimplemented. These undefined and unimplemented behaviors fill in the details with specialized subclasses. Hence, Inheritance can also help in the original conceptualization of a programming problem, and in the overall design of the program.

The **code reusability** helps in case 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 programming situations.

2.5 METHOD OVERRIDING AND POLYMORPHISM

A significant addition made to the capabilities of functions or methods in OOP is that of ***method overloading***. With this facility the programmer can have multiple methods with the same name but their functionality changes according to the situation. Suppose a programmer wants to have a method for calculating the absolute value of a numeric argument. Now this argument can be any numeric data type like integer, float etc. Since the functionality remains the same, there is no need to create many methods with different names for different numeric data types. The OOP overcomes this situation by allowing the programmer to create methods with the same name like *abs*. This is called **function overloading**. The compiler will automatically call the required method depending on the type of the argument.

Similarly **operator overloading** is one of the most fascinating features of OOP. For example, consider an addition operation performed by **'+'** operator. It shows a different behavior in different instances, i.e. different types of data. For two numbers, it will generate a sum. Three types may be integer or float.

Let us consider the situation where we want a method to behave in a different manner than the way it behaves in other classes. In such a case we can use the facility of ***method overriding*** for that specific class. For example, in our earlier example of the last section, bicycle superclass and subclasses, we have a mountain bike with an extra set of gears. In that case we would override the "change gears" method of that subclass (i.e. Mountain Bike) so that the rider could use those new gears.

Polymorphism

After classes and inheritance, polymorphism is the next essential feature of OOP languages. ***Polymorphism*** allows one name to be used for several related but slightly different purposes. The purpose of polymorphism is to let one name be used to specify a general class of action. *Method overloading* is one kind of polymorphism. We have already dealt with this type of polymorphism. The other type of polymorphism simplifies the syntax of performing the same operation with the hierarchy of classes. Thus a programmer can use polymorphism to keep the *interface* to the classes clean;

he doesn't have to define unique method names for similar operations on each derived class.

Suppose we have three different classes called *rectangle, circle and triangle* as shown in *Figure 12*. Each class contains a ***draw*** method to draw the relevant shape on the screen. When you call a ***draw*** method through a function call, the required ***draw*** will be executed depending on the class, i.e., rectangle, circle or triangle.

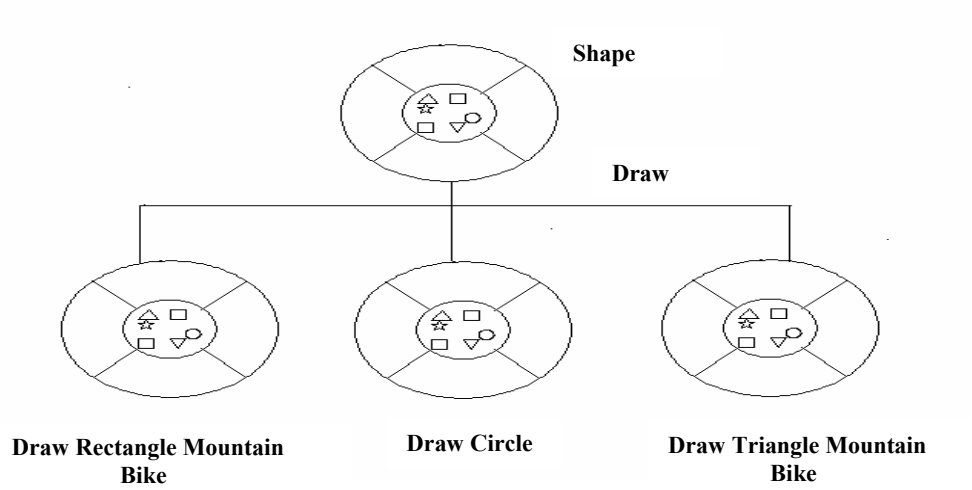


Figure 12: Polymorphism

Thus, Polymorphism plays an important role in allowing objects to have different internal structures (by method overloading or method overriding) but can share the same external interface.

☞ Check Your Progress 3

1) What are the benefits of inheritance?

……

……

……

2) Differentiate between method overloading and method overriding?

……

……

3) Explain message passing with an example.

……

……

2.6 SUMMARY

Object oriented programming takes a different approach to solving problems by using a program model that mirrors the real world problems. It allows you to easily decompose a problem into subgroups of related parts.

The most important feature of an object-oriented language is the object. An object is a logical entity containing *data* and *code* that manipulates that data. They are the basic run time entities. A class is a template that defines the variables and the methods common to all objects of a certain kind. Abstraction consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental properties. Classes use the concept of abstraction and are hence known as **Abstract data type**

(ADT). **Encapsulation** is the mechanism by which data and methods are bound together within the object definition. It is the most important feature of a class. The data is insulated from direct access by the program and it is known as data hiding. **Inheritance** is the process by which an object can acquire the properties of another object in the hierarchy. The concept of inheritance also provides the idea of reusability. New classes can be built from the existing classes. Most of the OOP languages support **polymorphism**. It allows one name to be used for several related but slightly different purposes. It is available in the form of method or operator overloading and method overriding. Polymorphism is extensively used in implementing inheritance.

2.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Every object in the world has its state and behavior. For example, a student object may have name, programme of study, semester etc. The variables name, programmes of a study, semester are known as instance variables. The value of instance variables at a particular time to decide the state of an object.

   Instance variables of different objects of the same class may have same value for example two different student may have same name, but mind it that they are not same.

2) An object is a software bundle of variables and related methods. Everything that the software object knows (state) and can do (behavior) is expressed by the variables and the methods within that object. In object-oriented software, it's also possible to have many objects of the same kind that share characteristics. Therefore, a class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.

3) Objects are instance of a class are combination of variables and methods used to represent state and behaviour of objects. One the class is defined its object can be used any where with the properties which are defined for it in class without modification

Check Your Progress 2

1) With data abstraction, data structures can be used without having to be concerned about the exact details of implementation. Classes act as abstract data types as they are defined as a set of abstract attributes and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created.

2) Two main benefits of encapsulation are:

   a) Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system.

   b) Information hiding: An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it.

3) Inheritance: It is the feature by which classes can be defined in terms of other classes. The class which feature is used in defining subclass is known as superclass. Subclasses are created by inheriting the state of super classes this can be seen as reasonability.

Inheritance is seen as generalization to specialization. In multiple inheritance attempt is made to define a specialized feature class which inherit the features of multiple classes simultaneously.

In multi level inheritence specialization is achieved step by step and the last class is the hierarchy is most specialized.

Check Your Progress 3

1) Inheritance offers the following benefits:

   a) Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times. Once a superclass is written and debugged, it need not be touched again but at the same time can be adapted to work in different situations. Reusing existing code saves time and money and increases a program's reliability.

   b) Programmers can implement superclasses called abstract classes that define "generic" behaviors. The abstract superclass defines and may partially implement the behavior, but much of the class is undefined and unimplemented. Other programmers fill in the details with specialized subclasses. Hence, Inheritance can also help in the original conceptualization of a programming problem, and in the overall design of the program.

2) With method overloading, programmer can have multiple methods with the same name but their functionality changes according to the situations. Whereas method overriding is used in the situation where we want a method with the same name to behave in a different manner in different classes in the hierarchy.

3) Message Passing – Object communicate with each other by message passing when object A want to get some information from object B then A pass a message to object B, and object B in turn give the information.

   Let us take one example, when student want to get certain information from office file. In this situation three objects student, clerk, and office will come in picture.

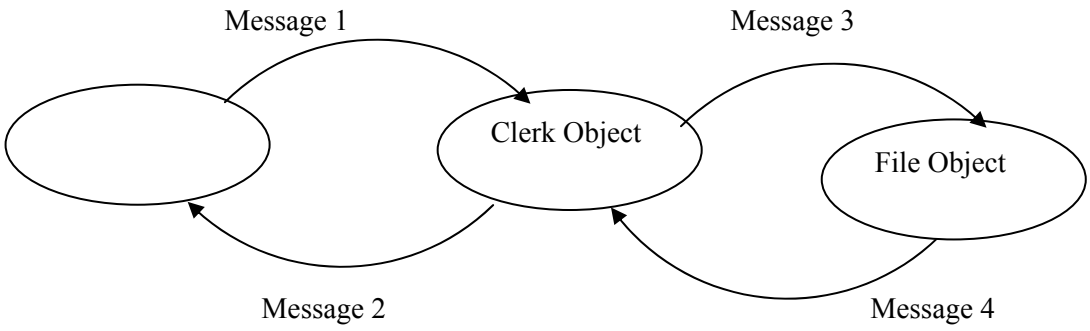


Figure 13: Message Passing